Automated Design of Application-Specific Smart Camera Architectures

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof.dr.ir. J.T. Fokkema,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op maandag 4 februari 2008 om 15:00 uur

door

Wouter Caarls

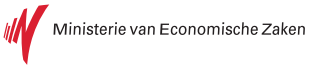
doctorandus in de Kunstmatige Intelligentie
geboren te Amsterdam

Dit proefschrift is goedgekeurd door de promotoren:
Prof.dr.ir. L.J. van Vliet
Prof.dr.ir. P.P. Jonker

Samenstelling promotiecommissie:

Rector Magnificus,	voorzitter
Prof.dr.ir. L.J. van Vliet,	Technische Universiteit Delft, promotor
Prof.dr.ir. P.P. Jonker,	Technische Universiteit Eindhoven
	Technische Universiteit Delft, promotor
Prof.dr.ir. H.J. Sips,	Technische Universiteit Delft
Prof.dr. H. Corporaal,	Technische Universiteit Eindhoven
Prof.dr.ir. H. Bal,	Vrije Universiteit Amsterdam
Dr. E. Visser,	Technische Universiteit Delft
Dr.ir. R. Kleihorst,	NXP Semiconductors
Prof.dr. I.T. Young,	Technische Universiteit Delft, reservelid

This work was supported by the Dutch government in their PROGRESS research program under project EES.5411, and carried out in the Quantitative Imaging group of the Imaging Science and Technology department in the faculty of Applied Sciences of the Delft University of Technology.



ISBN 978-90-9022688-0

Contents

Chapter 1

Introduction

Common wisdom dictates that having the right tool is half the job. The trouble is finding the right tool – or even being aware that it exists! – and learning how to use it. This holds double in computer science, where the number of tools practically exceeds the number of uses. However, often we stick to what we know, and use a programming language and processor architecture that we are familiar with. Only when it becomes apparent that our favorite tools are not sufficient do we start looking for other solutions, leading to costly redesign efforts.

Fortunately, the tools in computer science are much more flexible than those used in crafts. In computer science, if you know how to hammer in a nail, you only need an appropriate *compiler* to use the same algorithm for screwing in a screw. In the past, it has therefore been enough to know just one or a few general-purpose programming languages. But the increasing demands of high data throughput applications such as computer-assisted camera surveillance require radically different, parallel processing architectures. The large semantic gap between language and architecture makes programming akin to actually hammering in the screw.

The common approach to closing this gap is learning a new, architecture-specific language. However, this is not portable and therefore does not solve the redesign problem when the chosen architecture proves insufficient, nor does it help us finding the right architecture in the first place. We need to structurally expand a programmer's toolbox to include many current and future architectures, and – since we cannot expect the programmer to have the knowledge necessary to choose between architectures he does not know – automate the architecture selection.

Instead of bringing the language closer to the architecture, we propose to bring it closer to the *algorithm*, as the programmer certainly is intimately familiar with that. Such an *algorithm-specific language* (ASL) is easy to use because it caters to the specific needs of the algorithm, and it allows the program to run on any processor which implements its interface. Instead of choosing an architecture, the programmer now chooses an ASL to use, which can be done in a much more defined and informed manner.

As the program is now architecture-independent, we can automatically select an appropriate architecture *after* developing the application. The result is a processing architecture tailored to the application: the right tool for the job.

1.1 Smart cameras

The drive for our research is the field of embedded image processing. Even though the miniaturization and associated performance increase of microprocessors has been phenomenal, analyzing images delivered by current sensors at video speed (thirty frames per second) is still a daunting task. Specialized parallel architectures, such as single-instruction multiple-data (SIMD) processors, are necessary to maintain the required performance at an acceptable level of power consumption.

It is important to note that we are not targeting multimedia tasks such as video compression and image enhancement, which are static and often implemented in application-specific integrated circuits (ASICs) or vendor-supplied libraries. We are interested in the extraction and analysis of information from the images, which can then be used to make control decisions.

A device which integrates an image sensor with a processing architecture for analyzing the images is called a Smart Camera (SMARTCAM). The output of the processing is often not an image but image features, measurements, or control decisions. Such a device must be able to handle low-level pixel processing (noise removal, edge detection, segmentation) as well as feature extraction and high-level decision making. These tasks place vastly different requirements on the processor architectures implementing them. Consequently, there is no single architecture that is best in all situations, and an efficient SmartCam solution will have to be a heterogeneous multiprocessor.

Parallel programming is notoriously hard, even more so if the processors are heterogeneous. Heterogeneous multiprocessors are therefore usually not part of a programmer's toolbox, even if they offer the most efficient solution to the problem at hand. Algorithm-specific languages allow us to add any such architecture to the toolbox, and to automatically determine the optimal configuration.

1.2 Algorithm-specific languages

Architecture-independent programming is not a new concept; general-purpose languages such as C and Java can be compiled and executed on a large number of processors. Unmodified, however, their use is limited to sequential processors. Although there have been many attempts to automatically translate a C program to a parallel architecture, they are invariably inefficient, or efficient only for a small subset of C programs.

Once we realize that there is no reliably efficient way to translate a general-purpose sequential language to a parallel implementation, it makes sense to look at special-purpose sequential languages. By specializing a language to just those concepts necessary for the implementation of a particular class of algorithms, we accomplish two things: the language becomes easier to use, and can be efficiently translated to more (parallel) architectures.

The efficiency of the translation depends on the size of the language subset that a particular algorithm requires. Simple algorithms which use only a small subset can be translated very efficiently and can run on very simple processors. Complex algorithms using large subsets will generally be able to exploit less parallelism and can only run on more flexible processors.

We must therefore provide a range of algorithm-specific languages, each for a different class of algorithms, and each providing a different trade-off between generality and efficiency. If the user then chooses the most restrictive language in which his algorithm can be specified, it will execute the most efficiently and on the largest range of processors.

Of course, an application consists not of a single algorithm, but of many different algorithms connected together. Each algorithm may use its own language; this provides the ultimate freedom and flexibility to *map* the application onto an architecture, as it imposes the least number of requirements on the individual architecture components. In turn, this leads to efficient execution on heterogeneous multiprocessor systems.

1.3 Automated design space exploration

Choosing the right heterogeneous multiprocessor architecture for an application is not an easy task. Apart from choosing the characteristics of the individual processors, we need to determine how many of each processor type to take and how they are to be connected. Each architecture constitutes a trade-off between the objective variables: performance (speed/latency), power consumption and chip area.

Again, there is no single best architecture, even for a single application. Which architecture is more desirable depends on the relative priorities of the objective variables. Since these are hard to specify beforehand, we need to present the user with a set of *optimal trade-offs* (Pareto points) among which he can make the final decision himself.

It is of course impossible to actually execute the program on all architectures in the design space. In the first place, we will need to *simulate* the application, as the hardware is not available during the design phase. To reduce simulation time, we use the fact that our model of computation splits the program into separate algorithms which can be simulated individually.

Secondly, the design space is too big to explore using a brute-force method. We use a multivariate heuristic technique to limit the search space while still providing a reasonable approximation of the optimal trade-offs. By repeatedly choosing an architecture, simulating it, and refining the choice, we move towards successively better approximations. Because the design space is highly irregular, the heuristic that we use is based on *genetic algorithms*, since they do not assume a smooth objective space.

The design space is further limited by the use of an *architectural template*. The template contains the modes in which the architecture is allowed to change, such as word size, type and number of execution units, memory organization, interconnection, etc. Each architecture is an instantiation of this template.

1.4 Design flow

We can now construct the central design flow of our framework, illustrated in figure 1.1. A programmer starts by selecting the appropriate languages and us-

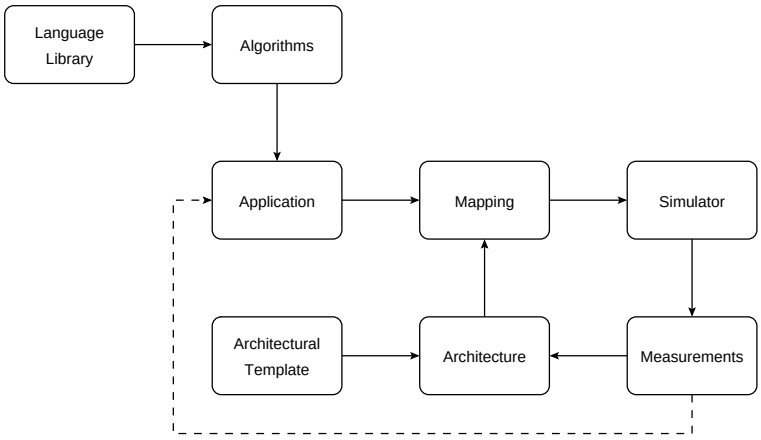


Figure 1.1: Design flow for the development of SmartCam applications. The dotted line denotes the manual restructuring of code if none of the presented architectures are sufficient.

ing these to construct his algorithms. The algorithms are connected to create an application, which is then mapped onto an instantiation of the architectural template. Simulating the application provides performance measures that can be used to refine the architecture.

It is possible to *incrementally* restructure an existing application to use our framework. In this case, the programmer starts by only rewriting the most computationally intensive inner loops of the application. If the trade-offs found by the design space exploration are not to his liking, he can rewrite more parts of the application.

1.5 Contributions and thesis outline

Chapter 2 of this thesis gives an introduction to the field of embedded image processing, its typical applications, processor architectures, algorithms and programming languages. We present a set of algorithm classes and match those to architectural features using a processor taxonomy. This supports the conclusion that there is no single best processing architecture, and that we need architecture-independent programs to be able to efficiently exploit all the options.

Chapter 3 presents the novel concept of algorithm-specific languages by relating them to functional programming and algorithmic skeletons. It then introduces stream programming as a way of connecting the algorithms, and explains how algorithm-specific languages can be seen as a generalization of the kernel languages used in stream programming.

Chapter 4 contains details about our implementation of algorithm-specific languages. We have created a new meta-programming language especially designed for the source-to-source translation of C-like languages into (parallel) C derivatives. This allows advanced programmers to add support for new algorithm classes or processor architectures. The language uses a novel technique which we call *pseudo-*

dynamic meta-programming to blur the distinction between the meta-level and source-level parts of the code.

In chapter 5 we discuss the execution of an architecture-independent program on a multiprocessor architecture, especially the problem of mapping operations to processors under dynamically changing conditions. We introduce a new performance prediction technique that incorporates both task cooperation and task dependencies, and present results on the efficiency of our approach.

Chapter 6 details how the architecture independence of an application program and properties of the streaming model of computation can be used to efficiently explore the design space of possible processing architectures using Pareto optimization of performance, energy and area. To our knowledge, such a complete integration of the trajectory from source code to a suitable parallel heterogeneous architecture has not been shown before. Results are presented on the convergence and coverage of the exploration, using simulated robotic soccer and augmented reality case studies.

Finally, in chapter 7 we summarize our work and discuss the results.

Chapter 2

Embedded image processing

Embedded systems are special-purpose computer systems designed to perform a dedicated function. Often, this function includes measuring a property of the environment and acting upon it. For example, a home thermostat measures the air temperature in a room and controls the gas valve and water flow in a central heating system. Such control functions have *real-time constraints*, meaning that an action has to be taken within a certain period after the change in the environment.

In embedded image processing applications, this period is often in the order of 10-100 milliseconds (though it depends on the properties of the dynamic system that is being controlled). In this time frame, many algorithms dealing with noise reduction, segmentation, feature extraction and decision making have to be performed. Since video sequences contain vast amounts of data, processing all the information takes a lot of computing power, often more than is available in regular microprocessors.

In terms of power consumption and speed, an application-specific integrated circuit (ASIC) solution will always be the most efficient. However, the non-recurring engineering (NRE) cost is very high, because on the one hand the design is time-consuming, and on the other the cost of making a leading-edge lithographic mask set already exceeds a million US dollars [125][1]. Both of these problems are only likely to grow in the future, as designs are getting larger and features sizes smaller [67].

The high NRE costs make ASICs unsuitable for small production runs. Furthermore, they make it uneconomical to modify the implementation based on new conditions or algorithmic insights. Finally, if an application consists of many different possible operations, only a few of which are active at one time, an ASIC requires hardware to be available for all operations, leading to high area overhead.

We are interested in applications for which a single general-purpose CPU is not fast enough – or too inefficient in terms of power consumption –, and which are too specialized, unstable, or dynamic for ASIC implementation. We will describe common image processing algorithms used in such applications, analyze what requirements they put on the processor architecture implementing them, and discuss

[1] Multi-project wafers or limited-custom masks such as sea-of-gates designs can bring this figure down, at the expense of per-unit cost.

the languages that may be used to program those architectures.

2.1 Application domain

Many current image processing applications, especially those in machine vision tasks such as industrial inspection, operate in a well-controlled, static environment. Because the environment does not change (except for the properties that have to be measured), they can manage with a static succession of algorithms. These applications are most suitable for an ASIC implementation.

In other situations, the task of the vision system depends on the environment. A photocopier might select different algorithms depending on whether the input is color or black-and-white, text or graphics. A digital video recorder needs to either compress or decompress video depending on whether it is recording or playing back. Still, such *use cases* are themselves static, and switching between them is sporadic. However, performance demands may still require the use of multiple processing devices, each specialized for a certain use case.

The most interesting application domain, and the one we will focus on, requires dynamic reactions to the environment on a frame-by-frame basis. This can be limited to just the front-end vision (such as adjusting noise reduction algorithms based on the signal to noise ratio and scene content), but can also change the entire control behavior of the system [11, 91]. Often, the dynamic environment also leads to many algorithmic changes over the lifetime of the system, thereby requiring a programmable solution.

It may be clear that dealing with this dynamic behavior is mostly a matter of being able to quickly change the way in which the algorithms inside an application are configured and interconnected. Apart from that, however, the fact that the environment is dynamic also has an impact on the types of algorithms that are used. Most importantly, the algorithms should be robust against variable lighting conditions, occlusions, motion, and other factors not under the control of the system.

Note that guaranteeing a reaction within a certain timeframe is hampered by the dynamic task connections. We will not address such guarantees in this study, and use the performance constraints as a guideline only. Because devices that operate in a dynamic environment are often untethered, energy consumption is typically constrained to hundreds of milliwatts to a few watts.

2.2 Processor architectures

Having restricted our scope to applications requiring programmable solutions and robust algorithms, there is still a very large range of processors to choose from. The main factors that exert an influence on the decision (apart from cost) are the speed at which the device can execute the algorithms, and the amount of energy it needs for the execution.

Speed and efficiency are largely a function of the amount of parallelism that can be exploited, as increasing the clock speed of a single processor is limited by heat production and power dissipation [51]. This parallelism comes in many different

flavors, such as bit-level (doing an addition on 8 bits at a time), instruction-level (executing an addition and a multiplication simultaneously), data-level (multiplying many different values at the same time) and task-level (convolving one image while segmenting another). All these need to be exploited for optimally efficient execution.

Section 2.3 will analyze the requirements of common image processing algorithms for efficient execution on a parallel processor architecture. Because we are investigating single algorithms, we will ignore task-level parallelism (that is the subject of section 2.5). We will also ignore bit-level parallelism, because we will assume all architectures allow the concurrent processing of entire bytes or words. This leaves us with instruction and data-level parallelism, for which we will need a consistent description of architectural support in the form of a taxonomy.

2.2.1 Taxonomy of parallel computing

The most common taxonomy for (parallel) computation is that by Flynn [50]. Flynn distinguishes four classes of computers:

- Single instruction stream - single data stream (SISD, sequential computation)
- Single instruction stream - multiple data stream (SIMD, vector computing)
- Multiple instruction stream - single data stream (MISD, not generally used)
- Multiple instruction stream - multiple data stream (MIMD, cluster computing)

This taxonomy makes essentially two distinctions: whether there are one or more instruction sequencers, and whether an instruction addresses multiple memories or only one. The main problem of this system is that it is not detailed enough for our purposes – having only two classes for parallel computers (SIMD and MIMD).

Whether an algorithm can be efficiently executed in parallel depends on many more factors. One important aspect is the degree of local autonomy in the processors, which can take a number of different forms [54, 83]. Essentially, these constitute different dimensions, spanning a space that encompasses everything from truly centralized (SIMD) to truly distributed (MIMD) autonomy. The most efficient execution architecture for a certain algorithm, then, is the one which distributes only as many resources as is necessary to exploit the parallelism inherent to the algorithm, while centralizing the rest. We will consider the following dimensions:

- *Number of processing elements (PEs).* This determines how many instructions may be executed in parallel.
- *Homogeneous or heterogeneous processing elements.* This specifies whether all processing elements are the same or not. There may be differences in instruction set, frequency, memory, access to external resources, etc.

- *Local or global instruction sequencing.* This specifies whether each processing element may branch independently; local instruction sequencing means there are multiple instruction streams.
- *Instruction synchronization.* Even with only one instruction stream (global instruction sequencing), processing elements may execute different parts of the stream at different times (buffered execution). Conversely, locally sequenced processing elements will include "barrier" instructions which are executed concurrently by all elements.
- *Shared or distributed memory.* If each processing element has its own memory, the memory bandwidth is higher. On the other hand, accessing non-local memory becomes more difficult.
- *Local or global memory address generation* (per-processor indirect memory addressing). Local memory address generation with global instruction sequencing means that, while every processor executes the same *load*, they may read from *different* parts of memory. Shared memory always implies local memory address generation.
- *Interconnect organization.* This describes how the processing elements are connected. It has a large impact on the communications bandwidth. Popular choices are rings, meshes, and crossbars. Often, a separate broadcast capability is also included.
- *Local or global communication address generation.* Analogous to memory address generation, a communication operation may require each processor to communicate with the same relative address (such as their left neighbor in a ring interconnect), or it may compute the address locally.
- *Communication latency.* Tightly coupled systems can communicate with single cycle latency. Shared memory coupled systems typically require tens to hundreds of cycles. Finally, networked systems need many tens of thousands of cycles.

Figure 2.1 illustrates how these components interact in a generalized parallel system.

2.2.2 Examples

Table 2.1 categorizes a number of contemporary processing architectures in terms of our taxonomy. While superscalar processors such as the Intel Core 2 microarchitecture (Intel Corporation, USA) are not generally considered to be a parallel system, they exploit a considerable amount of instruction-level parallelism through wide execution paths.

Furthermore, the nodes of the DAS2-TUD (Advanced School for Computing and Imaging, The Netherlands) cluster themselves consist of Intel Pentium processors, while the IMAP-CE (NEC Corporation, Japan) processing elements contain multiple (heterogeneous) function units. The TriMedia (Philips Semiconductors, The Netherlands) contains instructions which allow the bytes in a 32-bit data

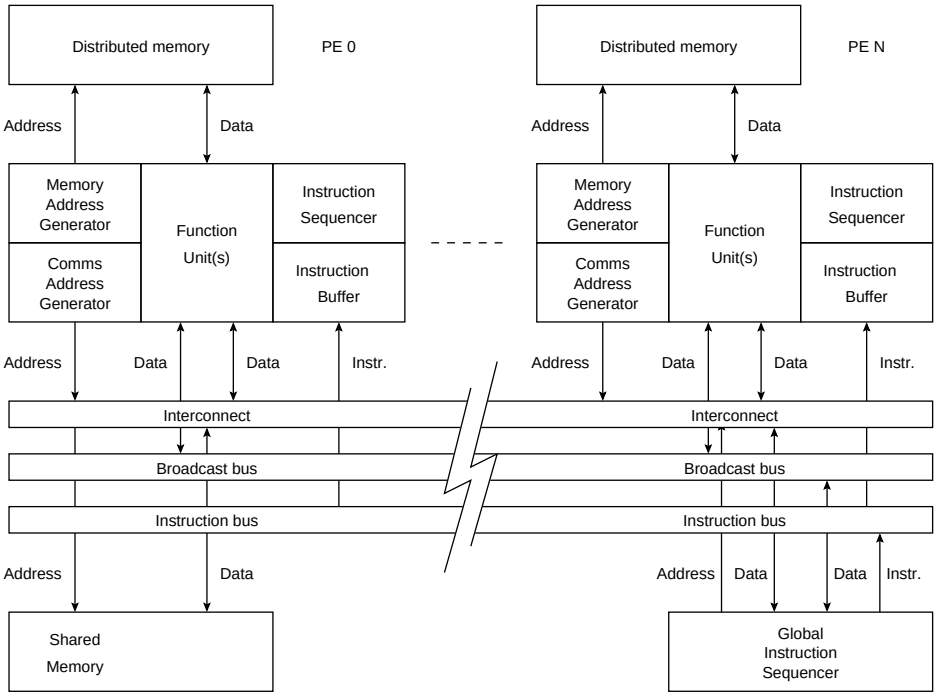


Figure 2.1: Generalized view of a parallel system. Many components can be either present or absent (such as the local address generators) or may have different properties (such as the interconnect or the function units).

word to be treated as single values, offering a limited form of data parallelism. The taxonomy could therefore apply to multiple levels of abstraction. For each architecture, we will only consider the level that exposes the largest degree of parallelism.

Intel Core 2 microarchitecture

The Core 2 [135] is Intel's latest microarchitecture to implement the x86 instruction set. It is a 5-issue out-of-order superscalar processor. This means that while the instruction stream is one dimensional, different parts of it are executed by different execution units, not necessarily in the order in which they appear in the stream; each execution unit has its own instruction queue. The execution units are heterogeneous (for example, one can only do loads from memory), and can forward results using a fully connected bypass network.

Intel produces chips with multiple of these cores on a single die, in a shared memory configuration. The number of cores per die is expected to grow in the future.

Philips TriMedia

The TriMedia [118] is a 5-issue VLIW processor. This is a different approach from superscalars but achieves the same result: a number of heterogeneous execution units is fed from a single instruction stream. In VLIWs, however, the instructions are scheduled by the compiler instead of the processor: each instruction in the stream contains a sub instruction for every execution unit, and these are executed in lock-step, reducing hardware complexity and power dissipation.

The main drawbacks of this approach are the large code side and the inefficiency of branching instructions.

Philips XeTaL

XeTaL (Philips Research Laboratories, The Netherlands [1]) is an SIMD processor specifically targeted to near-sensor processing such as fixed pattern noise correction and color reconstruction, but due to its fully programmable nature it can also be used for tasks like segmentation and stereo vision (through block matching between a left and a right camera image). It contains 320 10-bit fixed-point PEs with little local autonomy and small memories (16 image lines at a horizontal resolution of 640 pixels). Notably, the PEs have no indirect addressing capability, and only one of the operands of a multiplication may be local. The PEs are connected in a line (broken ring), and have access to a broadcast bus.

The control processor has a 12-bit integer ALU and 30 registers. The program memory is limited to 1024 instructions.

NEC IMAP-CE

The IMAP-CE [85] SIMD processor is similar to the XeTaL but has less, more powerful, PEs. Each of its 128 8-bit PEs is a 4-way VLIW with 24 registers and 2KB of local memory. The control processor is 16 bit, with 26 registers, 2KB local

data memory and 32KB program memory. An external 256MB SDRAM can be used if more memory is required. Data from this memory can be copied under DMA.

The IMAP-CE's PEs are interconnected using a ring. For binary operations, it supports *neighborhood parallelism* by gathering the data of the 8-connected neighborhood of a pixel in a single cycle. Each PE may generate its own memory address, but not the communication address.

A redesign of the IMAP-CE, called the IMAPCAR, uses 16-bit instead of 8-bit PEs.

TU/e DC-SIMD

The DC-SIMD [49] architecture is a prototype linear SIMD array that brings instruction buffering to SIMD processors with local communication address generation. Instruction buffering was introduced to avoid the long delays associated with dynamic communication in other architectures [68, 16]. In this case, execution is not fully synchronous, as each processor may wait a different amount of time depending on communication distance. If these distances are evenly distributed (so that a processor waiting for a long-distance communication is later likely to require only a short waiting time), DC-SIMD is faster than repeated shifting.

NVidia G80

Graphics Processing Units (GPUs) used for 3D visualization have become progressively more capable of scientific computing. Their processing elements started as configurable fixed-function units specifically for pixel (*fragment*) and triangle (*vertex*) operations. Later, they became programmable, and the G80 (NVidia Corporation, USA [40]) is the first such architecture to provide homogeneous scalar processing elements, unifying the fragment and vertex processors.

The G80 actually consists of 16 SIMD processors with 8 processing elements each. Instead of memory local to the PEs each processor has 16KB of *shared* memory, divided in 16 banks. As long as there are no bank conflicts, memory access is as fast as using registers. This is not the case for access to *global* memory, which has a 200-300 clock cycle latency. To hide this latency, a *thread scheduler* can switch to a different task while waiting for the transaction to complete. Using global memory is the only way for the processors to communicate.

ASCI DAS2-TUD

MIMD systems created by connecting commercial off-the-shelf components (*Beowulf* clusters, [121]) have become very popular in the last decade. The Distributed ASCI Supercomputer 2 (DAS2) consists of five such clusters, of which DAS2-TUD is one. Each of its 32 nodes contains 2 1 GHz Intel Pentium III processors with 1GB shared memory, and they are connected using Myrinet-2000 (Myricom, Inc., USA). Myrinet-2000 allows any permutation of connections between nodes to communicate bidirectionally at full 2Gb/s bandwidth using a Clos network. The minimum communication latency is around 10.000 clock cycles.

An new cluster, called DAS3, has now been installed. The TUD site contains 68 2.4 GHz AMD Opteron (Advanced Micro Devices, USA) processors connected by 1Gbps Ethernet.

2.3 Algorithms[2]

Image processing is a very large field, even if we restrict ourselves to embedded and real-time applications. It is therefore impossible to analyze all algorithms. We will, however, present a number of popular algorithms in this sub domain. Of course, what is possible in real-time is a shifting target as processors become faster.

In our analyses, we will always assume an optimal distribution of the image(s) over the available processors, and enough memory to store all relevant information. The following notation is used:

- X, Y and Z are digital images, treated as partial functions which map a pixel location to some value: $X : \mathbb{N}^2 \rightarrow \mathcal{R}(X)$. The domain of X, that is, all pixel locations for which X is defined, is denoted by $\mathcal{D}(X)$. Images are assumed to be square, having $|\mathcal{D}(X)|$ pixels, where ${}^2log\sqrt{|\mathcal{D}(X)|} \in \mathbb{N}$. The range (possible pixel values) is denoted by $\mathcal{R}(X)$, and can be an intensity, a Cartesian product of intensities (in the case of *tensor* images), or any other value.
- $\mathbf{p}$, $\mathbf{q}$ and $\mathbf{r}$ are pixel locations. $X_{\mathbf{p}}$ is the value of the pixel in X at location $\mathbf{p}$.
- S and T are sets of relative pixel locations, used as a neighborhood or *structuring element* around a pixel.
- R is a set of absolute pixel locations.
- f and g are functions.
- C is an array of constants.

2.3.1 Parallelism

We will classify the considered algorithms into categories which offer the same amount of parallelism. On an algorithmic level, we are only interested in the *inherent* parallelism of the algorithm, that is, the (average) amount of primitive operations that may be executed in parallel assuming infinite resources. This can be determined by counting the number of computational steps in the algorithm, and dividing it by the number of steps a hypothetical infinitely parallel machine would need to execute it.

Two operations may not be executed in parallel if, under transitive closure, there exists a dependency between them. There are three kinds of dependencies which affect the inherent parallelism of an algorithm [14]:

[2]The basis of this section was formed during an internship at NEC, Japan in May 2005

Table 2.1: Characterization of different parallel processing architectures according to the taxonomy of section 2.2.1

Characteristic	**Core 2**	**TriMedia**	**XeTaL-1**	**IMAP-CE**	**DC-SIMD**	**G80**	**DAS2-TUD**
PEs	5	5	320	128	320	128	24
Heterogeneous	Y	Y	N	N	N	N	N
Instruction sequencing	Global	Global	Global	Global	Global	Clustered per 8 PEs	Local
Synchronization	Buffered	Lockstep	Lockstep	Lockstep	Buffered	Clustered per 8 PEs	Free
Memory organization	Shared	Shared	Distributed	Distributed	Distributed	Shared per cluster	Distributed
Memory address generation	Global	Global	Global	Local	Local	Local	Local
Interconnect organization	Full	Full	Ring	Ring	Ring	Using global shared memory	(8, 8, 8)-Clos
Comm address generation	Global	Global	Global	Global	Local	Local	Local
Communication latency (cycles)	1	1	1	1	1	≈100	≈10.000

- *Flow dependencies* arise when an operation writes a variable which is read by an operation occurring later in the algorithm description.
- *Anti dependencies* are those where an operation writes a variable which is read by an operation *preceding* it in the algorithm description.
- *Output dependencies* occur when two operations write to the same variable.

Often, the algorithm can be rewritten to avoid anti and output dependencies, but flow dependencies are really inherent. Whether the inherent parallelism is *exploitable* depends on the amount of available processing elements, their capabilities and interconnections. In short, it depends on whether the pattern in which the algorithm accesses the pixels is supported by the hardware. We will therefore define our categories based on these access patterns.

2.3.2 Low-level operations

Low-level operations work on entire images. They take images as input and produce images as output. This means that essentially all pixels in the input image will be visited by the algorithm.

Pixel to pixel operations

$$\forall \mathbf{p} \in \mathcal{D}(Y) : Y_{\mathbf{p}} \leftarrow f(X_{\mathbf{p}}) \tag{2.1}$$

Pixel operations, such as binarization and addition, contain trivial parallelism. Each pixel may be processed completely in isolation, resulting in parallelism $|\mathcal{D}(Y)|$. Depending on f, it may be possible to exploit instruction-level parallelism within each pixel. For example, computing an arctangent is often implemented as a multi-linear approximation, some parts of which may be executed in parallel.

Anisotropic pixel operations

$$\forall \mathbf{p} \in \mathcal{D}(Y) : Y_{\mathbf{p}} \leftarrow f(\mathbf{p}) \tag{2.2}$$

Anisotropic operations have access to the pixel coordinates. These are often used to generate images for use in later operations, such as ramps and subsampling maps. Apart from requiring each processing element to know which pixel locations it is processing, this does not affect the available parallelism.

Pixel lookup operations

$$\forall \mathbf{p} \in \mathcal{D}(Y) : Y_{\mathbf{p}} \leftarrow f(X_{\mathbf{p}}, Z) \tag{2.3}$$

Lookup operations can access a lookup table (Z) to determine the value of a pixel. This category includes such operations as color mapping and segmentation, but if X is a displacement map and Z an image, it may also be used for lens distortion correction. The inherent parallelism is still $|\mathcal{D}(Y)|$, but implementations may suffer from read contention on Z. If the added latency due to such accesses can not be sufficiently amortized through duplication, pipelining, or instruction buffering, this may severely limit the exploitable parallelism.

Lookup operations require local memory address generation for efficient implementation. Additionally, 2D lookups (such as displacement maps) require local communication address generation.

Pixel to global operations

$$\forall \mathbf{p} \in \mathcal{D}(X) \forall \mathbf{q} \in g(X_{\mathbf{p}}) : Y_{\mathbf{q}} \leftarrow f(X_{\mathbf{p}}, \mathbf{q}) \tag{2.4}$$

This is the dual of the pixel lookup class, generating multiple output pixels per input pixel. Clearly, this suffers from output dependencies if the sets of output pixels generated by g are not disjoint. Even if the sets are disjoint, write contention on Y limits exploitable parallelism; efficient implementations therefore require the memory to be distributed according to g. In addition, pixel to global operations have the same address generation requirements as pixel lookup operations. An example is the Hough transform, where each pixel of an image generates a curve in Hough space.

Neighborhood to pixel operations

$$\forall \mathbf{p} \in \mathcal{D}(Y) : Y_{\mathbf{p}} \leftarrow f(\{X_{\mathbf{p}+\mathbf{q}} | q \in S\}) \tag{2.5}$$

Any neighborhood operation, from mathematical morphology to convolution. Again, without further knowledge of f, inherent parallelism is still $|\mathcal{D}(Y)|$. However, if S is not contiguous (for example, because the convolution kernel contains many zeroes), exploiting this parallelism requires a denser interconnect than rings or meshes.

Recursive neighborhood to pixel operations

$$\forall p \in \mathcal{D}(Y) : Y_{\mathbf{p}} \leftarrow f(\{X_{\mathbf{p}+\mathbf{q}} | q \in S\}, \{Y_{\mathbf{p}+\mathbf{r}} | r \in T\}) \tag{2.6}$$

A recursive operation allows access to certain parts T of the *output* image. This introduces flow dependencies, and therefore an ordering and limitations on parallelism. T might even be chosen such that no legal order can be found. In the most popular recursive neighborhood operations, such as distance transforms [18], T is the neighborhood that *would be available assuming row-major iteration* (iterating over columns in the inner loop). Parallelism is limited to $\frac{\sqrt{|\mathcal{D}(Y)|}}{2}$, because at every step only a diagonal line of the output image can be calculated. Full exploitation requires at least a tightly coupled ring connected system with local memory address generation. Loosely coupled systems will lose too much time waiting for the processed boundary values on which f depends.

Bucket processing

$$\begin{aligned} &Y \leftarrow X \\ &Bucket \leftarrow R \\ &\text{while } \exists \mathbf{p} \in Bucket \\ &\qquad Bucket \leftarrow (Bucket \setminus \mathbf{p}) \cup g(Y_{\mathbf{p}}) \\ &\qquad Y_{\mathbf{p}} \leftarrow f(Y_{\mathbf{p}}) \end{aligned} \tag{2.7}$$

A bucket is initialized with a number of *seed points* (R). For each seed point, the output is updated using f, while new points can be added using g. There is no explicit ordering on the treatment of the bucket elements. It may be implemented as a *stack*, in which case the order is depth-first. An example operation which can be implemented using bucket processing is the binary propagation of a seed over a mask image, shown in figure 2.2.

By distributing the bucket over the available processing elements [100], each PE reading from the bucket that contains the pixels it has in local memory, we attain a parallel implementation. The inherent parallelism is limited by the size of the bucket, which itself depends on g, and is thus data-dependent. Exploitable parallelism is further limited by the process of writing remote buckets. For buckets sizes larger than one, local memory address generation is necessary.

Ordered-iteration bucket processing

$$
\begin{aligned}
&Y \leftarrow X \\
&Stack \leftarrow R \\
&\text{while } Stack \neq \emptyset : \\
&\quad \forall p \in Stack : \\
&\qquad Y_{\mathbf{p}} \leftarrow f(\{X_{\mathbf{p+q}} | q \in S\}) \\
&\qquad NewStack \leftarrow Stack \cup g(\{X_{\mathbf{p+q}} | q \in S\}) \\
&\quad Stack \leftarrow NewStack \\
&\quad X \leftarrow Y
\end{aligned}
\tag{2.8}
$$

Many wavefront propagation algorithms, such as skeletonization, require the neighborhood of a pixel in order to determine the output value and update the bucket. Anti-dependencies (where an updated value is erroneously read as part of the neighborhood) must therefore be resolved by creating a temporary output image and placing a partial ordering on which pixels may be processed: all pixels in one iteration must be completed before starting the next.

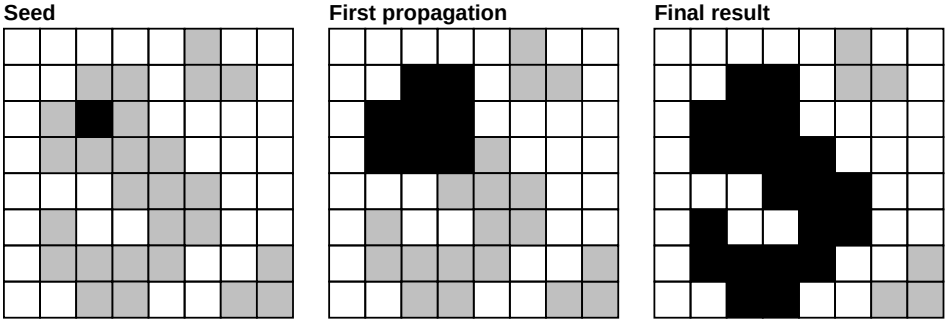


Figure 2.2: Successive stages of the binary propagation of a seed over a mask. The first figure is the seed R; the middle figure illustrates the pixels that may be reached by the first application of g, while the right figure is the final image.

Iteration ordering further limits parallelism (although, again, data-dependently), while neighborhood addressing has the same requirements as in eq. 2.5.

2.3.3 Intermediate-level operations

Intermediate-level operations reduce the amount of data in an image, either by selecting pixels or objects from an image, or by accumulating the pixels into a scalar or vector by some statistical operation.

Scalar reduction

$$\forall \mathbf{p} \in \mathcal{D}(X) : y \leftarrow f(X_{\mathbf{p}}, y) \tag{2.9}$$

Reduces an entire stream to a scalar value. As we have not defined an *order* on X, f must be both associative and commutative. Many reduction operations, such as maximum, minimum and addition, satisfy these conditions. From commutativity also follows that $\mathcal{R}(X) = \mathcal{R}(y)$. Such operations may be performed in ${}^2\log|\mathcal{D}(X)|$ steps using a *reduction tree*, leading to inherent parallelism $\frac{|\mathcal{D}(X)|-1}{{}^2log|\mathcal{D}(X)|}$.

Full exploitation of this parallelism requires a tightly coupled interconnect which can emulate a tree (such as full or hypercube interconnect). More restricted forms, such as rings or meshes, have to spend time shifting the intermediate values. A broadcast bus is then needed to avoid too many shifts at the upper levels of the tree. In that case, ring parallelism is in the order of $\frac{1}{3}|\mathcal{D}(X)|^{\frac{2}{3}}$ (see footnote[3]).

Vector reduction

$$\forall \mathbf{p} \in \mathcal{D}(X) : Y_{X_{\mathbf{p}}} \leftarrow f(Z_{\mathbf{p}}, Y_{X_{\mathbf{p}}}) \tag{2.10}$$

Vector reductions are used to aggregate values for a number of different items simultaneously, the most well-known being histogramming, with $f = (+)$ and $Z = 1$; in other cases, X is often an image of *object indices*. The amount of work, and the inherent parallelism, is the same as scalar reduction. Full exploitation is still possible, but now requires $|\mathcal{D}(Y)|$ $(= |\mathcal{R}(X)|)$ parallel reductions.

A naive ring implementation using the same method as for scalar reduction requires $|\mathcal{D}(Y)|$ times more work, since all elements of Y have to be combined at each node of the reduction tree. From a certain size of $|\mathcal{D}(Y)|$ onwards, it becomes advantageous to distribute Y over the available processors (with each processor creating a local version and combining them afterwards), leading to parallelism $\frac{1}{2}|\mathcal{D}(X)|^{\frac{1}{2}}$ for $|\mathcal{D}(Y)| \leq \sqrt{|\mathcal{D}(X)|}$.

Filtering

$$Y = \{X_{\mathbf{p}} | Z_{\mathbf{p}} = \mathsf{true}\} \tag{2.11}$$

[3] This may be achieved in three phases: first, reducing the values local to each processor. Next, doing parallel reductions by shifting the data over increasing distances. Finally, a sequential reduction by gathering the remaining data over the broadcast bus. Each phase takes $|\mathcal{D}(X)|^{\frac{1}{3}}$ steps.

Generates an output containing fewer elements than the input, such as in subsampling. There are no dependencies, and inherent parallelism is $|\mathcal{D}(X)|$. Exploitation efficiency depends on the time it takes to redistribute Y.

Contour following

$$\begin{aligned}
&\forall \mathbf{r} \in R : Y_{\mathbf{r}} \leftarrow (0, \mathbf{r}) \\
&\text{while } R \neq \emptyset : \\
&\quad \forall \mathbf{r} \in R : \\
&\qquad \mathbf{p} \leftarrow \mathrm{Snd}(Y_{\mathbf{r}}) \\
&\qquad Y_{\mathbf{r}} \leftarrow (Y_{\mathbf{r}}, \mathbf{p} + f(\{X_{\mathbf{p}+\mathbf{s}} | \mathbf{s} \in S\})
\end{aligned} \tag{2.12}$$

This is similar to bucket processing, except that instead of updating the image, for each seed a list of visited points is kept. In 2.12, this list is constructed as a left-recursive tuple (Snd returns the right member of the tuple). Each contour has to be followed sequentially, but different contours may be followed in parallel. Inherent parallelism is therefore $|R|$.

2.3.4 High-level operations

High-level operations work on the features extracted from images by intermediate-level operations, and generate new features, or *decisions* based on those features. The data structures involved in these calculations are often not arrays, but lists or trees. In many cases, efficient parallelization requires a different algorithm than the sequential case.

Often, high-level algorithms are not specific to image processing, but rather draw from fields such as linear algebra, optimization, pattern recognition, and symbolic reasoning. We therefore only discuss two examples.

Sorting

The fastest sequential sorting algorithm in the average case is *quick sort*, requiring $O(NlogN)$ steps. For linear SIMD arrays, the theoretically optimal worst case is $O(N)$, because a value may have to be shifted from one end of the array to the other. This theoretical optimum is reached by *odd-even transposition sort* (which is a parallel version of *bubble sort*). On meshes, *shear sort* [113] can reach $O(\sqrt{N}logN)$ by alternatively sorting rows and columns (although this requires limited local communication address generation), and *bitonic sort* [13] reaches $O(log^2 N)$ on hypercubes.

Machines with large communication latencies (such as MIMD clusters) will require $N >> P$ for efficiency. In that case, each processor usually keeps a sorted list. An algorithm that is often used in this case is *sample sort* [60], with time complexity $O(\frac{N}{P} log \frac{N}{P})$.

Branch and bound search

Branch and bound search is common in optimization problems such as path finding and machine learning. The concept is to start with a partial solution and move

towards the final goal, expanding upon that partial solution which minimizes a certain *cost function*. Examples are depth-first search (evaluate first expanded solution first), uniform cost search (expand solution with minimum cost) and A* (expand solution with minimum incurred + predicted cost).

Because of its formulation, B&B search is sequential: only one partial solution is expanded at a time. However, we may expand more than one in parallel, provided that the final solution is the same as in sequential expansion. Each processor keeps its own priority queue of expanded but unevaluated partial solutions, and the tops of these queues are regularly exchanged. Note that this means that non-essential partial solutions are expanded, making it difficult to asses the inherent parallelism.

The exchange of partial solutions can be done locally between neighbors [41, 140], and updating a local priority queue only requires local memory address generation. However, how a partial solution is to be expanded depends on the solution, as does the number of children it generates. This makes B&B algorithms less suitable for implementation on machines without local instruction sequencing [79]. In addition, the size of the priority queues will generally exceed the local memory of SIMD processing elements.

2.3.5 Discussion

Image processing uses a wide variety of algorithms, displaying a large diversity in the memory access patterns and inherent parallelism. How this parallelism may be exploited depends on the architecture on which the algorithm is executed; each architecture will require its own implementation in order to achieve the best speedup.

Furthermore, which architecture is most suited to exploit the parallelism depends on the algorithm in question. While an architecture with a large amount of local autonomy and dense interconnect will support all the modes presented in this section, the cost of these features will dramatically reduce the number of processing elements that can be implemented, leading to a loss of exploitable parallelism for less complex operations.

There is therefore no single optimal image processing architecture, but rather a continuum of architectures, each of which is optimal for only a certain *class* of algorithms.

2.4 Languages

We will now present a number of APIs and languages that can be used to program parallel devices. In order to restrict our scope somewhat, we only describe C-based languages, and have selected one representative from each of four categories targeted at different architectures and applications. As in section 2.3, we will focus on the execution of single algorithms.

2.4.1 Shared memory: OpenMP

OpenMP [45] is a multi-platform application program interface (API) for shared-memory parallel programming in C/C++ and Fortran. It consists of a set of compiler directives (*pragmas*) for expressing fork-join parallelism and a library of utility functions. Fork-join parallelism is characterized by a *master thread* which executes the sequential parts of the program, and splits off *worker threads* in the parallel sections. Once the parallel section is finished, the worker threads are destroyed or suspended until the next parallel section.

Program 2.1 Example of the OpenMP `parallel for` directive.

```
  int y, x;
  double out[height][width], in[height][width];

#pragma omp parallel for private(x)
  for (y=1; y < height-1; y++)
    for (x=0; x < width; x++)
      out[y][x] = in[y-1][x] - in[y+1][x];
```

The most common form of parallelism is the data parallelism encoded in `for`-loops. In program 2.1, a vertical gradient is calculated in parallel for all rows in an image. Depending on the number of available processors, OpenMP spawns a thread for each block of `y` values. Unless stated otherwise, all variables are shared. Since each thread should have its own copy of `x`, it is declared `private`.

Other forms of parallelism are also possible, such as master-slave data parallelism or task parallelism. Program 2.2 illustrates how task parallelism is expressed: all the code in the block following the `omp parallel` directive is usually replicated over the threads, but the `sections` directive restricts this by only spawning a thread to execute each `section`. Therefore, `gauss_dx` and `gauss_dy` are executed in parallel.

Program 2.2 Task parallel gradient magnitude in OpenMP. The Gaussian derivatives `gauss_dx` and `gauss_dy` are executed in parallel.

```
void gradmag(image_t *in, image_t *out)
{
  image_t dx, dy

#pragma omp parallel sections
  {
  #pragma omp section
    gauss_dx(in, &dx);
  #pragma omp section
    gauss_dy(in, &dy);
  }

  norm(&dx, &dy, out);
}
```

Additional directives exist to express reductions, and to define *critical sections*, which may only be executed by one thread at a time. Library functions allow interaction with the OpenMP run-time system, such as adjusting the number of worker threads, and querying the current thread identifier.

A key aspect of shared memory parallel programming systems is the absence of a *data distribution* mechanism. As each processor may access any part of the memory with the same latency, the location is unimportant except for caching. While efforts have been made to implement OpenMP for distributed-memory systems [47], the lack of control over the data distribution can lead to a performance loss compared to true distributed-memory programming environments.

OpenMP may be used for *incremental parallelization*. Because OpenMP is a strict superset of the host language, any normal sequential program is a valid OpenMP program. Furthermore, OpenMP directives may be added one at a time, locally, without restructuring the rest of the program. This enables the programmer to direct his parallelization efforts to only those parts of the program where the benefit is largest.

2.4.2 MIMD clusters: MPI

The Message Passing Interface (MPI, [53]) is the industry standard for parallel scientific computing. It is based on the single-program multiple-data (SPMD) paradigm, where the same program runs on all processors of an MIMD cluster, but each processor operates on different data elements and may follow a different execution path. MPI is available for both C/C++ and Fortran.

MPI provides a library of communication routines that implement *message passing*. This is an inherently distributed-memory approach, where no variables are shared unless explicitly communicated by sending messages. The communication primitives include both point-to-point and collective operations.

Program 2.3 illustrates this programming paradigm. The same program is run on each processor, but they receive different values for `rank` (third line). These are then used to process different parts of the image; each processor is allocated `lines` image lines, plus one line of border both above and below (see figure 2.3). After the local computation, a collective communication operation (`MPI_Allreduce`) is used to determine whether any pixels changed. Finally, the borders are exchanged, first shifting up, and then down (`MPI_Sendrecv`).

Unlike in OpenMP, it is not possible to incrementally parallelize a sequential application. Data distribution is under explicit imperative program control, and must be carefully thought out before writing a parallel program. While this creates a significant hurdle to writing parallel code, the level of control that is possible allows for efficient implementations.

Because MPI is so low-level, and because it is available for so many architectures, it is often used to implement higher-level parallel languages or libraries. It can be seen as the "assembly language" of parallel programming, abstracting over different cluster interconnects.

Program 2.3 MPI program for iterating a neighborhood operation until idempotence. Each processor is allocated a series of image lines to process, and the image borders are exchanged during each iteration.

```
MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
lines = HEIGHT/nprocs;

while (1)
{
  int res, changed=0;

  for (y=1; y < lines+1; y++)
  {
    /* Code to process local image stripe */
    /* Set changed to 1 if any pixel changed */
  }

  MPI_Allreduce(&changed, &res, 1, MPI_INT, MPI_SUM,
                MPI_COMM_WORLD);
  if (!res) break;

  MPI_Sendrecv( &img[1][0], WIDTH, MPI_INT, prev(rank, nprocs), 0,
                &img[lines+1][0], WIDTH, MPI_INT, next(rank, nprocs), 0,
                MPI_COMM_WORLD, &status);
  MPI_Sendrecv( &img[lines][0], WIDTH, MPI_INT, next(rank, nprocs), 0,
                &img[0][0], WIDTH, MPI_INT, prev(rank, nprocs), 0,
                MPI_COMM_WORLD, &status);
}
```

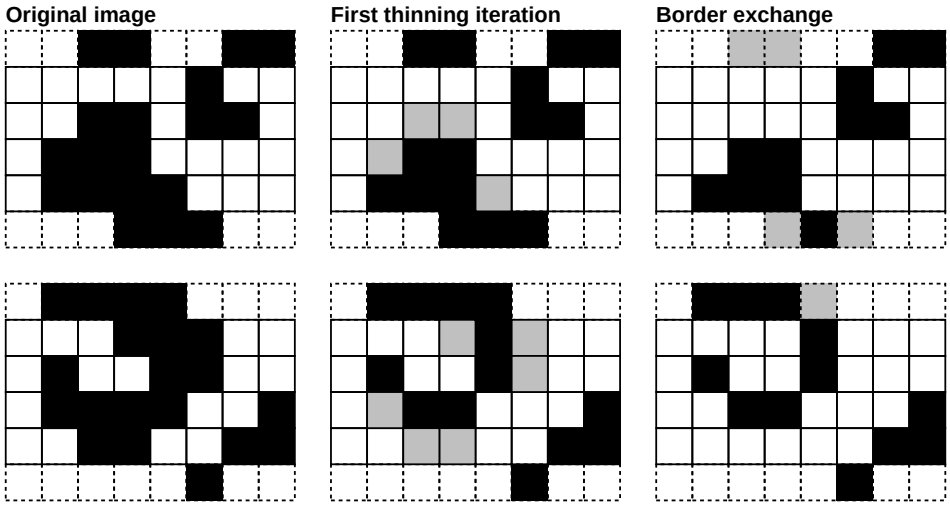


Figure 2.3: Border exchange for 2 processors during an iterative thinning algorithm. We assume a *cyclic* vertical dimension, so that the upper neighbor of the top image line is the bottom image line. Grey values indicate changed pixels; the overlapping borders (dotted lines) are not updated until the communication step.

2.4.3 SIMD arrays: 1DC

Both MPI and OpenMP are intended for use on MIMD machines; MPI for distributed-memory systems, and OpenMP for shared-memory ones. They are essentially control-oriented: the system (or programmer) distributes loop iterations. SIMD languages, on the other hand, are data oriented. By introducing *parallel data types*, they expose parallelism while still maintaining a single thread of control.

1DC [86] is an extension of the C language that was designed for use with the NEC IMAP-Vision SIMD processor [55] (the predecessor of the IMAP-CE). It adds a new keyword "`separate`" to declare that a variable should be distributed over all processors; calculations involving such a variable are done by all processors at the same time. There are also a number of new operations on these variables for accessing neighboring values.

Program 2.4 illustrates some of these operations. It implements bucket processing in the spirit of eq 2.7 using stacks. The `:||` *global or* operator is used to check whether any of the stacks are nonzero (this requires a broadcast bus), and the `:<` and `:>` operators access the right and left neighbor respectively. 1DC also provides an `:[i:]` indexing operator to access the value of a single processing element.

New control constructs are introduced to conditionally enable or disable PEs: the code following `mif` is only executed by those PEs for which the condition is true. This may sometimes require a rethinking of the intended algorithm. For example, in program 2.4 it would be natural to add the left neighbor of a processed pixel to the stack. However, in 1DC it is not possible to write something to a

Program 2.4 1DC code for the stack-based 4-connected propagation of `255` (the seed) over `1` (the mask). Each processor keeps a separate stack, and iteration continues until all stacks are empty. Note the use of indirect addressing to deal with differing stack sizes. This code assumes the border to be zero.

```
#define push(img, y, stk, sp)\
        mif (img[y] == 1) { img[y] = 255; stk[sp++] = y; }

void propagate(separate uchar *img)
{
  separate uchar stk[HEIGHT];
  separate int sp = 0;
  int i;

  for (i=0; i < HEIGHT; i++)
    mif (img[i] == 255) stk[sp++] = i;

  while (:||sp) /* Any stack is nonempty */
  {
    separate int y = 0;

    mif (sp)
    {
      y = stk[--sp];

      push(img, y-1, stk, sp); /* Pixels above and below */
      push(img, y+1, stk, sp);
    }

    push(img, y:<1, stk, sp); /* Pixels left and right */
    push(img, y:>1, stk, sp);
  }
}
```

neighboring pixel. Instead, each PE adds the pixel left to the one processed by his right neighbor to his own stack.

1DC (and other SIMD languages, such as XTC [98] and HPF [52]) combine the power of distributed data types with the familiarity of a single thread of control, although care must be taken as to the proper distribution. They vary mainly in the generality of the added operations, and the strictness with which they enforce the distribution. For example, XTC does not allow indirect addressing, while HPF allows non-local access as though the data were local (remote memory accesses will simply incur a penalty).

2.4.4 Hardware compilation: Handel-C

Handel-C was developed to provide a familiar environment for creating synchronous FPGA hardware designs. Instead of using a hardware description language such as VHDL [9] or Verilog [63], the programmer writes a C-like program, which is then compiled to the logic gate level. Handel-C is a derivative of the Occam [70] language, which was itself heavily influenced by Hoare's CSP [69].

All expressions in Handel-C are created using combinatorial logic, and may not have side effects; the cycle time of the design therefore depends on the most complex expression in the program. Assignments to variables take one cycle. Parallelism is introduced using the `par` construct, which executes all its statements at the same time. Variables in such constructs may not be written by more than one statement. As assignment takes one cycle, reading a variable in the same `par` construct as it is written to refers to its *previous* value.

Different parallel processes may communicate using FIFO *channels* (`chan` keyword). This is the only reliable way of communications for statements in different `par` constructs. "`!`" is the write operator, while "`?`" reads a value from the channel.

Handel-C also adds a host of features dealing with the bit-widths of integer variables, the placement of arrays (in RAMs or registers), RAM accesses, hardware interfaces, clocking, signals and general issues of targeting a program to a specific device.

Program 2.5 implements a streaming 3x3 convolution. Each cycle a pixel is read from `in`, and a result is written to `out`. The code is pipelined to avoid too many gate delays, so the output value is delayed for a number of cycles. The line buffers are `WIDTH-3` bytes long. An illustrative diagram of the resulting hardware can be found in figure 2.4.

It can be seen from this example that Handel-C, although having a C-like syntax, requires a clear vision of what the generated hardware is going to be and a thorough understanding of the parallel semantics to be able to write efficient programs. Especially the explicit pipelining needed to create a fast design calls for a change of perspective.

2.4.5 Discussion

While all the languages we discussed are based on C syntax and imperative semantics, they differ in the kind of parallelism that may be exploited. Even though

Program 2.5 Pipelined 3x3 convolution in Handel-C. The `do` loop produces one result every cycle, but the convolution of a particular pixel is only available 4 cycles after the bottom-right pixel of its neighborhood is read.

```
macro proc convolve (chan int in, chan int out, int mask[3][3],
                     chan int linebuffer[2][2])
{
  int img[3][3], tmp[3][3], row[3], acc;

  do
  {
    par
    {
      img[0][0]        = img[0][1];
      img[0][1]        = img[0][2];
      linebuffer[0][1] ? img[0][2];
      linebuffer[0][0] ! img[1][0];
      img[1][0]        = img[1][1];
      img[1][1]        = img[1][2];
      linebuffer[1][1] ? img[1][2];
      linebuffer[1][0] ! img[2][0];
      img[2][0]        = img[2][1];
      img[2][1]        = img[2][2];
      in               ? img[2][2];

      par (ii=0; ii < 3; ii++)
      {
        par (jj=0; jj < 3; jj++)
          tmp[ii][jj] = img[ii][jj] * mask[ii][jj];

        row[ii] = tmp[ii][0] + tmp[ii][1] + tmp[ii][2];
      }

      acc = row[0] + row[1] + row[2];

      out ! acc;
    }
  } while (1);
}
```

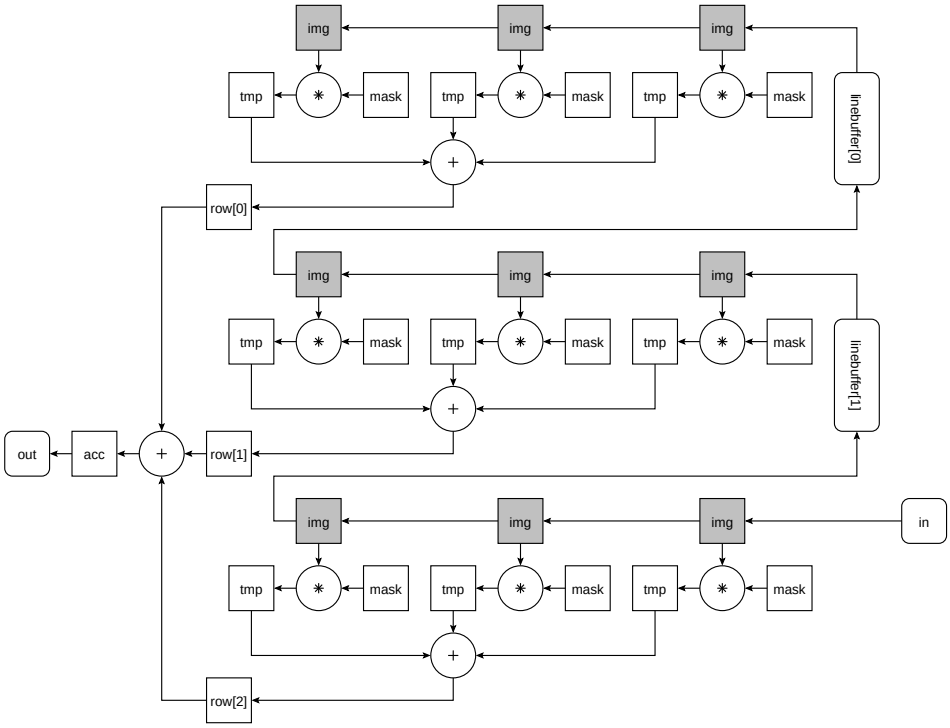


Figure 2.4: Hardware structure generated from the Handel-C code in program 2.5. Each square box is a register, and each circle is an expression. The registers make sure that the execution is pipelined. The gray registers contain the image.

many are retargetable (even 1DC has been implemented for the MMX and SSE SIMD extensions used in desktop processors), each has a specific type of architecture for which it is most suited, and also places firm restrictions on which hardware features must be present. Apart from this, we must deal with the fact that processor vendors generally support only one or at most a few languages.

Analogous to the discussion in section 2.3.5, it is difficult to choose a single language for implementing image processing algorithms. The choice is dependent on the architecture, which in turn depends on the specific algorithm.

2.5 Constructing applications

A typical application consists of many of the algorithms described in section 2.3, connected in various ways. Often, the basic access patterns are first combined to yield composite operations. For example, creating a run-length encoding (RLE) of a binary image may be achieved by first replacing the pixels that are different from their left neighbor by their linear index, filtering them out, and finally replacing all elements with the difference to their left neighbor, creating a list of run lengths. See figure 2.5.

As described in section 2.3.5, there is no single processor architecture on which all of these algorithms can be optimally implemented. This means that if the application is diverse enough (which is nearly always the case), it is beneficial to look at architectures containing more than one type of processor. From section 2.4.5 we may infer that this also requires the use of multiple languages.

2.5.1 Heterogeneous multiprocessing

In order to make efficient use of a multiprocessor system, all the processors must be kept busy with meaningful work (as opposed to waiting for data, or bookkeeping). In a heterogeneous system, where each processor has different strengths and weaknesses, this implies the exploitation of task parallelism. Note, however, that in typical image processing applications, the amount of exploitable data parallelism far outweighs the gains reachable by task parallelism. In our view, task parallelism is therefore purely a way to make efficient use of a heterogeneous system, and does not scale to more than a few (data-parallel) processors.

The main issue with the construction of task parallel heterogeneous systems is one of interfacing. These interfaces may be arbitrarily complex. For example, we might envision an SIMD and superscalar cooperating using shared memory and locking, as is the case with the IMAP-CE PCI board. This allows intricate cooperation but is therefore also difficult to code for and debug, especially if different languages are involved.

In signal processing systems, it is common to abstract these interfaces into a *process network*. Each sub algorithm is considered a black-box process, and communicates with other processes over *channels*. The properties of the channels and the rules to which the processes must adhere depend on the particular model of computation (MoC) that is being used. The application developer is now only concerned with a *software interface* that may be implemented in any number of ways.

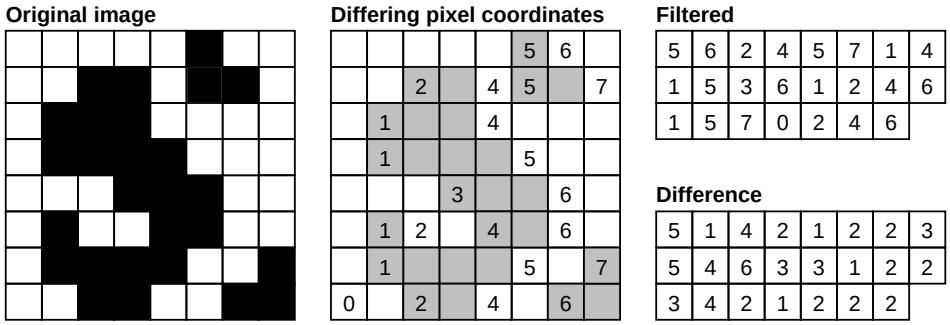


Figure 2.5: Run-length encoding as a combination of an anisotropic pixel operation (finding the index), neighborhood to pixel operation (placing the index in the pixels which are different from their left neighbor), filtering, and another neighborhood to pixel operation (replacing the X coordinates with differences). Indices are shown in X coordinates for simplicity.

We have already stated in section 2.1 that our application domain requires fast dynamic reconfiguration based on changes in the environment. In the context of a process network, this means being able to change the sources and destinations of the channels. Typically, such changes do not affect the entire process network, and in order to maintain sufficient task parallelism we would like to continue the execution during reconfigurations.

2.6 Discussion

We have shown that there are many processing architectures available for embedded image processing applications. However, not all of these architectures are equally suited for all applications. Depending on which classes of algorithms are used, and which requirements are put on factors such as performance, power consumption and cost, different processors – or combinations of processors – are most appropriate.

It may be clear that it is not always possible to choose the correct architecture beforehand. However, we have seen that the language in which an application is written already restricts the class of architectures which may be efficiently used. Therefore, if we wish to make an unbiased decision about the most suitable architecture, the application cannot be written in any of the languages we discussed.

In this entire chapter we have deliberately made a distinction between algorithms (which exploit data parallelism) and applications (which treat algorithms as black boxes that need to be connected). The reason being that it allowed us to discuss the different properties of each class of algorithms separately. From this point of view, it is not a huge leap to think about using languages which are dependent on the properties of the *algorithm* we wish to implement instead of the *architecture* on which it will be executed, thereby achieving architecture independence.

The next chapter will further detail this concept, and the rest of the thesis

discusses an implementation and how it can be used to make an unbiased decision about the most appropriate architecture for executing a certain application.

Chapter 3

Designing architecture-independent applications

There is a wide diversity of embedded image processing architectures. As was discussed in chapter 2, none of these architectures is superior to all the others. Which architecture is the most desirable is highly dependent on the application. In current practice, therefore, the choice of architecture is generally based on an estimate of the performance needed by the application.

However, a large bias is introduced against multiprocessor architectures and processors that are not programmed in a general-purpose language, because of the specialized knowledge that is necessary to effectively exploit their power. Furthermore, if the performance estimate proves incorrect, porting the application to a new architecture (especially if it consists of multiple processors) can be very time-consuming.

In order to overcome the latter problem, we aim to automatically determine the optimal architecture *after* the program has been written. As such, the application needs to be written in an architecture-independent way, since the architecture is unknown at design-time. As an additional benefit, the programmer does not need specialized architectural knowledge, because this would break the architecture independence. This removes the bias against multiprocessor systems.

We will describe how architecture independence can be achieved by using *algorithm-specific languages* (ASLs), where each algorithm in an application can be described with a different language. This requires the algorithms to be separated from each other, leading to a natural implementation on heterogeneous multiprocessor systems. We will relate our approach to *algorithmic skeletons* and *stream programming*.

3.1 Architecture independence through algorithm dependence

One of the main benefits of general-purpose languages such as C and Java has been their independence of the instruction set architecture (ISA) of a microprocessor. C achieves this by *compiling* the program to different architectures, while Java goes a step further by compiling to a *virtual machine* architecture, which is then emulated. Compilation to a virtual machine has the benefit of binary compatibility, but this is rarely an issue in embedded systems.

The architecture independence provided by these general-purpose languages is limited. First, they can only be compiled to architectures which observe a set of minimum requirements, especially on the available addressing modes. More importantly, however, they do not support *data distribution*, and can only be *efficiently* compiled to a restricted set of architectures that match their execution model.

That execution model is sequential, with variables holding intermediate results and control flow constructs such as conditionals and loops regulating a single flow of execution. Programs written with such an execution model in mind tend to have many dependencies, and for an efficient parallel implementation the algorithm itself must be changed. The analysis required by this step will remain beyond the capability of compilers for the foreseeable future. Furthermore, indirect addressing, pointer aliasing and global variables require that the compiler make pessimistic assumptions about the content of variables and indices to guarantee that the program has the same result before and after transformation, which leads to inefficiencies.

3.1.1 Algorithm-specific languages

One solution to this problem is to avoid the need for transformations from sequential to parallel code by letting the user write explicitly parallel programs, leading to the languages described in the previous chapter. As we noted before, though, such languages are architecture-class dependent. A better solution, therefore, is to restrict the programming language in such a way that the assumptions can be less pessimistic.

Of course, restricting a programming language reduces the possible functions it can compute. To maintain generality, we must allow *different* sets of restrictions, based on the operation the user is trying to implement. Each set of restrictions defines a different *algorithm-specific language (ASL)*, suitable for only a certain class of algorithms. Simple algorithms – those that can be specified in a very restricted language – can then be translated very efficiently and run on very simple and highly parallel processors. More complex algorithms using less restrictive languages will generally be able to exploit less parallelism and can only run on more flexible processors.

We have thus replaced architecture dependence by algorithm dependence. This is beneficial because the programmer is concerned with algorithms, not with architectures. Since the ASL is tailored to a specific class of algorithms, implementing such algorithms is also easier than in a general-purpose language.

We have discussed a number of algorithmic classes in section 2.3. They restrict the access to data structures to a certain *pattern*, and define an implicit iteration over the data structure. While this implicit iteration is not a necessary property of algorithm-specific languages, it yields an easy parallelization strategy because the data distribution is independent of the program. A nice conceptual way of thinking about such an ASL is viewing it as a higher-order function. This is precisely the view taken in the field of *algorithmic skeletons*, where higher-order functions repeatedly apply an input function (also called *kernel*) to the elements of some data structure. While the match is not exact, we will continue to use the skeleton terminology, which we will introduce in section 3.2.

Using a different language for each algorithm implies that the algorithms are defined separately, and not in one loop nest. This makes communication between them explicit and encourages reusability. In fact, the program outside the algorithms is only concerned with setting up these connections and global control flow. Such a distinction between a *kernel language* and *coordination language* is an important aspect of *stream programming*, which we will discuss in section 3.3.

3.2 Algorithmic skeletons

Algorithmic skeletons [36, 107] originated from the field of functional programming. We will first briefly introduce functional programming and its notation, after which we will introduce the concept of algorithmic skeletons and how they relate to algorithm-specific languages.

3.2.1 Functional programming

Functional programming differs in two main ways from imperative programming. First, there are no stored variables, and all functions are therefore *pure*: their results depend only on their arguments. A program consists of one expression leading to one result (although this may be a list or other data structure). Second, and more important in our context, functions are *first-class objects*: they can be passed as arguments, returned, and manipulated just like other values such as integers.

Objects have a *type*. For example, in the Haskell [74] language, the type of 1 is *Num*, while the type of $'a'$ is *Char*. Formally:

$$\begin{array}{lll} 1 & :: & Num \\ 'a' & :: & Char. \end{array} \tag{3.1}$$

Functions also have types. *log*, which takes the natural logarithm of a number, has the type $Num \rightarrow Num$:

$$log :: Num \rightarrow Num, \tag{3.2}$$

meaning that it is a function which maps a number to another number. In Haskell, function application is written simply as sequencing, binding strongly from left to right. If we want to create a function that applies the logarithm twice, we can

define it as

$$\begin{array}{lcl} \mathit{twicelog} & :: & \mathit{Num} \rightarrow \mathit{Num} \\ \mathit{twicelog}\, x & = & \mathit{log}\,(\mathit{log}\, x). \end{array} \tag{3.3}$$

It is the power of functional programming, however, that we can define a *higher-order function* that applies *any* function twice:

$$\begin{array}{lcl} \mathit{twice} & :: & ((a \rightarrow a), a) \rightarrow a \\ \mathit{twice}\,(f, x) & = & f\,(f\, x) \\ \\ \mathit{twicelog}\, x & = & \mathit{twice}\,(\mathit{log}, x), \end{array} \tag{3.4}$$

where a is any data type. *twice* takes a tuple of two arguments: a function to be applied twice, and the value to which to apply it. It returns the result of the double application. Another definition takes advantage of the fact that we may also *return* functions:

$$\begin{array}{lcl} \mathit{twice'} & :: & (a \rightarrow a) \rightarrow (a \rightarrow a) \\ \mathit{twice'}\, f & = & \backslash x \rightarrow f\,(f\, x) \\ \\ \mathit{twicelog'} & = & \mathit{twice}\ \mathit{log}. \end{array} \tag{3.5}$$

twice' now returns a function that applies f twice to its input, and therefore *twicelog'* returns a function that applies a logarithm twice to its input. This creation of functions "on the fly" is the main difference between functional programming and C function pointers.

Two of the first programming languages to develop such a functional style of programming were APL [71] and LISP [95]. APL also introduced the concept of *array programming* (where functions typically operate on many values at once) later popularized by Matlab.

3.2.2 Separating structure from computation

Algorithmic skeletons are higher-order functions which take as argument a function that works on some elements of a data structure, and return a function that applies the input function repeatedly over the entire data structure.

One of the most well-known skeletons is the *map* skeleton, which returns a function that applies a function $f :: a \rightarrow b$ to all elements of a list:

$$\mathit{map} :: (a \rightarrow b) \rightarrow ([a] \rightarrow [b]), \tag{3.6}$$

where $[a]$ denotes a list of basic type a. The skeleton does not specify the *order* in which the application is to take place, only that f will be applied to all elements of the input. The restriction being that all applications of f are independent.

Another well-known skeleton is *reduce*, which returns a function that applies a binary associative function $g : (a, a) \rightarrow a$ to reduce a list to a single value:

$$\mathit{reduce} :: ((a, a) \rightarrow a) \rightarrow ([a] \rightarrow a). \tag{3.7}$$

Again, the order is not specified. Since g is associative, there are many possible *implementations*, each resulting in a different order. Three possibilities are shown in figure figure 3.1.

Because the ordering is implemented by the skeleton, and hidden from the user, algorithmic skeletons separate the structure of a computation from the computation itself. Many different operations can be implemented using a single skeleton, and a single skeleton can be implemented on many different (parallel) architectures.

3.2.3 Skeletons as ASLs

The main difference between an algorithmic skeleton and an algorithm-specific language is that the input function to an algorithmic skeleton is *opaque*: the skeleton can only *call* the input function. As a result, the input function must be specified in a language which is understood by the target architecture compiler.

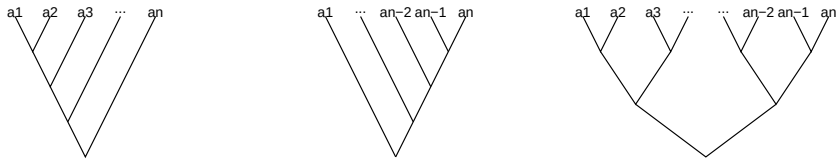
In contrast, an ASL specifies its own compiler, allowing the operations written in it to execute on many more target architectures. Naturally we cannot go about designing an entirely new programming language and compiler for every class of algorithms. Rather, we implement the compilation as a *source-to-source translation* of the input function into the final operation.

Consider the *map* skeleton of eq. 3.6. In a traditional functional program, the skeleton, its input and its output are all written in the same source language `S`:

$$map :: \left\langle \left\langle a \to b \right\rangle_{\mathtt{S}} \to \left\langle [a] \to [b] \right\rangle_{\mathtt{S}} \right\rangle_{\mathtt{S}}. \tag{3.8}$$

A traditional compiler translates any function in a source language `S` to a corresponding function in the target language `T`. This is usually done by first translating to a (possibly parallel) intermediate language `I`. The language in which the compiler is written is unimportant and therefore omitted:

$$\begin{aligned} &frontend && :: \left\langle f \right\rangle_{\mathtt{S}} \to \left\langle f \right\rangle_{\mathtt{I}} \\ &backend && :: \left\langle f \right\rangle_{\mathtt{I}} \to \left\langle f \right\rangle_{\mathtt{T}} \\ &compile && = backend\ .\ frontend. \end{aligned} \tag{3.9}$$



(a) Folded from the left (b) Folded from the right (c) Tree

Figure 3.1: Different orderings for a reduction. 3.1(c) provides the largest degree of parallelism.

In the above equation, (.) is the *function composition operator*, and should be read as "after." The compiler therefore first executes the *frontend*, followed by the *backend*.

As mentioned in section 3.1.1, it is necessary to limit the expressiveness of a sequential program in order to effectively exploit the parallelism in it. In this case, however, *frontend* is a general translator from `S` to `I`, and `S` itself is not limited; it is the kernel that is limited. To exploit parallelism *map* will therefore have to be a *specialized frontend*, translating from a subset of `S` to `I`:

$$\begin{aligned} &map && :: \langle a \to b \rangle_{\mathtt{S}} \to \langle [a] \to [b] \rangle_{\mathtt{I}} \\ &compile && = backend\ .\ map. \end{aligned} \tag{3.10}$$

To a lesser degree, the same argument holds for *backend*. While `I` may be annotated by the skeleton to communicate dependency information, it is often easier to write a specialized backend for each skeleton, thereby avoiding the need to define `I`. The choice depends on the relative difficulties of writing a number of specialized translators versus writing a single general translator.

We have chosen to have the skeletons translate directly to `T`:

$$map \quad :: \langle a \to b \rangle_{\mathtt{S}} \to \langle [a] \to [b] \rangle_{\mathtt{T}}. \tag{3.11}$$

Each skeleton is now a translator for an *implicitly parallel algorithm specific* subset $\mathtt{S_i}$ of `S`. Each skeleton will have multiple implementations, translating to different target languages $\mathtt{T_i}$, or run-time environments.

Because of backwards compatibility and adoption concerns, we would like the kernels to be written in C, or some language closely resembling C. Furthermore, our target processors are programmed in C or parallel C-derivatives. The skeletons are therefore mostly concerned with generating and manipulating C code. Chapter 4 describes the language in which the skeletons themselves are written.

3.3 Stream programming

In stream programming languages [120], an application consists of a number of *kernels* that work on *streams* of data *elements*, see figure 3.2. This promotes data locality, as it makes the communication explicit: the programmer has to explicitly connect the kernels together.

Because of this data locality, stream programs can be readily executed on heterogeneous distributed-memory systems by mapping the kernels to the available processors and the streams to the communication channels between them, see figure 3.3. This mapping is not part of the program, which is therefore architecture-independent.

Implementations of the stream programming concept differ in the way streams and kernels are defined and connected. We will discuss the most important possibilities below.

3.3.1 Streams

A stream is a series of data elements of a specific *type*. The type may be declared as a part of the stream declaration, or derived from the kernel which generates the

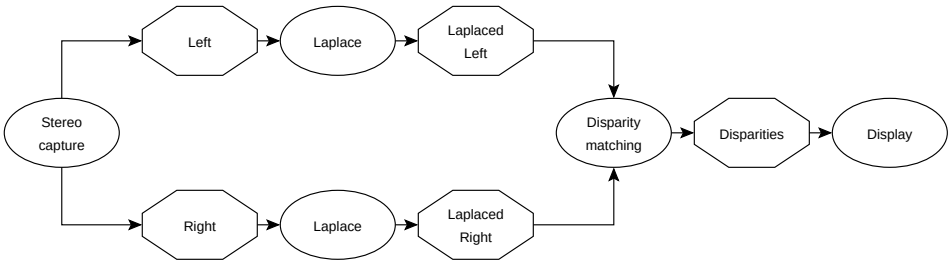


Figure 3.2: Stereo matching as a simple stream program. The left and right images are filtered and combined into a disparity image. The ellipses represent kernels, while the polygons denote streams.

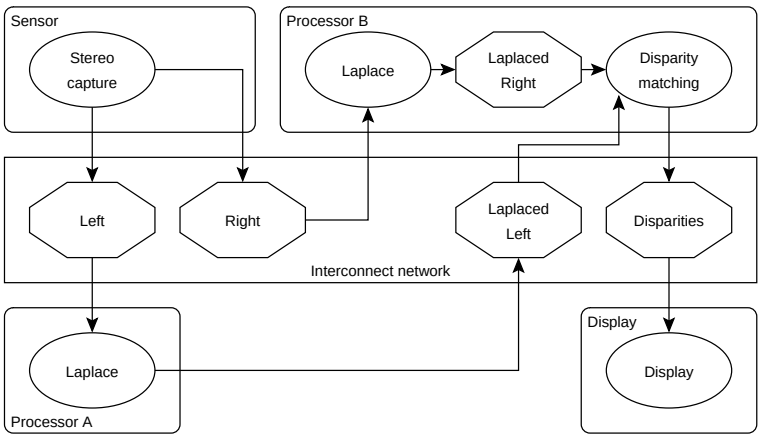


Figure 3.3: The stream program of figure 3.2 mapped onto an architecture. The kernels are mapped to processors, and the streams are mapped to communication resources (the interconnection network, or processor-local memories).

stream. The stream declaration may also include a fixed *length*. Such fixed-length streams mark a difference between two schools of thought.

In the first, all streams are bounded (although some may have variable bounds), implying that the kernels interacting with those streams have a *finite lifetime* determined by the stream lengths. In the other, conceptually infinite streams mean that a kernel's lifetime is infinite as well, unless limited by other means (usually referred to as *out-of-band* signalling).

A stream in the purest sense is one-dimensional, a strict series of elements. Since this may be too restrictive for dealing with multidimensional data, such as images, streams are sometimes attributed a *shape*. A shape imposes a spatial view of the stream, relating an element to its neighborhood.

In the Brook language [25], stream shapes are used in conjunction with *stencils* to provide access to a stream element's neighborhood. Stencilling creates a stream of overlapping neighborhoods that can be used to implement local neighborhood or finite difference operations (see figure 3.4). Stencil elements which would fall outside the stream are generated using a *border handling strategy* such as clamping or tiling. Program 3.1 shows the declaration and creation of a statically typed, fixed-length stream of overlapping 2D 3x3 neighborhoods.

Program 3.1 Stream stencils using Brook. `a` is a 2d stream of floats, and `b` is constructed as a stream of overlapping 3x3 neighborhoods of `a`. Each neighborhood ranges from top left to bottom right (X and Y range from `-1` to `1`), treating accesses outside of the stream's domain as if the stream were periodic (`STREAM_BOUNDS_PERIODIC`, also called tiling).

```
float a<256, 256>;
float b<256, 256>[3][3];

streamStencil(b, a, STREAM_BOUNDS_PERIODIC, 2, -1, 1, -1, 1);  .
```

We have distinguished three main stream properties: implicit vs explicit typing, finite vs infinite streams, and 1D vs multidimensional streams. For the highly dynamic image processing applications we are targeting, it is evident that finite streams with native support for multidimensional stream access are desirable. Finite streams allow us to decide which kernels to run for each image separately, and multidimensional stream access greatly simplifies writing local neighborhood operations.

In a language where stream connections are dynamic, implicit vs explicit typing

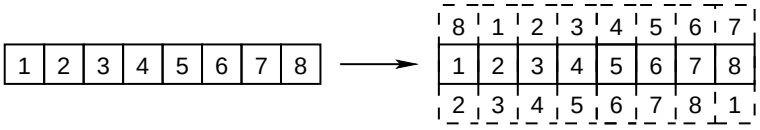


Figure 3.4: 1D stencilling. A stream of elements is transformed into a stream of neighborhoods.

is closely related to static vs dynamic typing. Statically typed programs are easier to analyze and can generate compile-time typing errors, while dynamically typed programs can be slightly easier to write.

3.3.2 Kernels

Kernels are functions that operate on streams. They read stream elements, do some computation, and write new stream elements. The most significant characteristic of a kernel language is whether the iteration over the stream elements is *implicit* or *explicit*. An implicit iteration strategy facilitates parallel execution, because there is no user-defined loop that can introduce false dependencies. On the other hand, the operation may be incompatible with the provided mode of iteration, such as a single loop over all elements.

Closely related to iteration is the concept of *state*. Explicit iteration naturally allows information from one iteration to be used in a successive one by using loop-carried variables (introducing dependencies). Implicit iteration can also support state, but this must be strictly regulated in order to avoid the pitfalls described in section 3.1 for automatic parallelization.

A third choice is between fixed or variable *rates* of production and consumption. If each iteration produces and consumes the same number of stream elements, it becomes possible to statically schedule multiple kernels, leading to efficient execution. Variable rates require dynamic scheduling techniques, which are inherently less efficient.

Program 3.2 illustrates explicit, stateful iteration in KernelC [94]. KernelC was specifically developed for the Imagine stream processor [78], which is an 8-way SIMD architecture; the `loop_stream` construct cyclically distributes the stream elements over the processing elements. The reduction after the loop is specific to an 8-way parallel architecture.

Program 3.2 A reduction in KernelC (from [94]) which calculates the sum of a stream. "`>>`" reads an element from the stream. Note that the between-PE summation of the total is specific to the 8-PE Imagine processor.

```
KERNEL sumStream(istream<int> in, uc<int> uc_total)
{
  int total = 0;
  loop_stream(in) {
    int ini;
    in >> ini;
    total = total + ini;
  }
  total = total + permute(LROTATE, total);
  total = total + permute(LROTATE2, total);
  total = total + permute(LROTATE4, total);
  uc_total = ucWrite(0, total);
}
```

Compare this to the implicit, fixed-rate stateless iteration of the Brook code

in program 3.3.

Program 3.3 Brook code for a 2d average filter. The stencilled input stream makes it easy to access the neighborhood, which is centered on index (1, 1).

```
void kernel average(float a<>[3][3], out float b<>)
{
  b = (a[0][0] + a[0][1] + a[0][2] +
       a[1][0] + a[1][1] + a[1][2] +
       a[2][0] + a[2][1] + a[2][2]) / 9;
}
```

It is clear that explicit iteration is the most general strategy. However, this also makes it the most difficult one to parallelize, as is shown by the awkward use of SIMD communication primitives in KernelC. Static rates are the easiest to analyze and schedule, but not all applications have static rates (run-length encoding is an example). Stateless kernels are the easiest to parallelize, but again, not all applications can be described using stateless kernels (infinite impulse response (IIR) filters can not, for example).

Based on these considerations it is not easy to decide upon the most desirable kernel language, because it depends on the algorithm. Skeletons and ASLs are therefore a natural solution, and we will expand on this in section 3.4.

3.3.3 Connecting kernels

A stream program consists of a number of kernels, connected using streams. These connections can be made by calling the kernels as functions that take streams as arguments. The stream connections are thus made under *imperative* program control. This means that they can depend on all variables that are available to the stream program, including data dependent ones.

Alternatively, the stream connections could be specified in a *declarative* way. A *stream graph* is constructed or specified once, and then allowed to execute; this combines well with conceptually infinite streams, as changing the connections once they have been set up is difficult (there is no "global control" to initiate such a change). A possible solution to this is event-based reconfiguration [99].

StreamIt [123] is an example of a declarative stream programming language. Program 3.4 shows how an iterative diffusion program can be written by combining a file reader, a number of diffusion steps, and a file writer in a pipeline. The streams in this code are implicit, because each stage of a StreamIt pipeline has only one input and one output. The data types of the streams are therefore also implicit, and derived from the kernels.

The declarative approach allows for straightforward whole-program optimizations, while these same optimizations require careful analyses before they can be applied in imperatively connected stream programs. In an application where reconfigurations are frequent and data dependent, declarative programming becomes

Program 3.4 Iterative diffusion in StreamIt. The `add` keyword adds a stage to the pipeline.

```
void→void pipeline IterativeDiffusion(int steps) {
  int i;

  add FileReader<float>("input.raw");
  for (i=0; i < steps; i++)
    add Diffusion();
  add FileWriter<float>("output.raw");
}
```

cumbersome however, as the concept of a stream graph loses much of its clarity when the processes can change on a frame by frame basis.

Our application domain has this property, and in such situations an imperative style is to be preferred. This means that optimizations must be restricted to those parts of the program that do not depend on run-time values. Often, this is further limited to program parts without branches or labels, called *basic blocks*.

3.3.4 Exploiting task parallelism

Stream computing can be used just for the exploitation of data parallelism by the kernels. In that case, each kernel is fully executed before starting the next one. If the stream program is to be mapped to multiple processors, though, they should efficiently make use of the between-kernel task parallelism that is inherent to their structure.

The most interesting form of task parallelism in this case is *pipelining*, where two tasks run concurrently on different parts of the same stream. This is also used on uniprocessors, in which case the stream is split into a number of strips, and each strip is fully processed by all kernels before the next strip is taken. Such *strip mining* reduces buffering and increases cache locality, saving memory bandwidth to external memory.

It is difficult to apply strip mining across data-dependent branches, however, since it is uncertain which kernels should be executed. This means that all kernels within a basic block must finish executing (and their data stored to external memory) before advancing to the next basic block, even if the data dependency is resolved before that time. This reduces task parallelism to zero at each branch.

A dynamic approach to pipelining, such as data flow execution [3], does not need static information about the kernels. In data flow execution, a kernel may run as soon as its input is available: the flow of data determines which kernels can be executed.

Strip mining, or another way of limiting memory usage, is clearly advantageous. In dynamic applications, however, it is imperative that execution can continue as much as possible over data-dependent branches. As data dependencies make the mixture of kernels that are running at any one time indeterministic, it is intuitive to schedule them dynamically based on the availability of their inputs, much like

data-flow execution.

Chapter 5 describes how we have implemented stream definition, dynamic connection, and the exploitation of task parallelism in an imperative environment using asynchronous RPC.

3.4 Stream kernels as skeleton inputs

Brook's stream kernels can be seen as functions that are passed to the *map* skeleton. Every element of the output stream is calculated separately from the others, only using the corresponding input element.

If the kernel has multiple input or output streams, we can first *zip* corresponding stream elements together into a stream of composite structures:

$$\begin{aligned} &zip :: [[a]] \rightarrow [[a]] \\ &zip \quad [[a_1, a_2, \ldots, a_n], [b_1, b_2, \ldots, b_n], \ldots, [z_1, z_2, \ldots, z_n]] = \\ &\qquad [[a_1, b_1, \ldots, z_1], [a_2, b_2, \ldots, z_2], \ldots, [a_n, b_n, \ldots, z_n]]. \end{aligned} \tag{3.12}$$

For kernels that need to access a neighborhood, we can *stencil* a stream to create a stream of neighborhoods:

$$\begin{aligned} &stencil :: (Int, [a]) \rightarrow [[a]] \\ &stencil \quad (k, [a_1, a_2, \ldots, a_n]) = \\ &\qquad [[a_1, a_2, \ldots, a_k], [a_2, a_3, \ldots a_{k+1}], \ldots, [a_{n-k+1}, a_{n-k+2}, \ldots, a_n]]. \end{aligned} \tag{3.13}$$

Brook also supports reductions, by marking an output with the `reduce` specifier. We can obtain the same behavior by first creating a stream of outputs using *map*, and then processing this stream using an appropriate instantiation of *reduce*.

Program 3.5 Brook code to calculate the sum of the squares of a stream.

```
void kernel sumsquares(float a<>, reduce float b)
{
  b = b + a*a;
}
```

The sum-of-squares Brook kernel in program 3.5 can be separated by first generating the sum-of-squares stream, and then performing a reduction on it:

$$\begin{aligned} &sumsquares \quad :: [a] \rightarrow a \\ &sumsquares \quad = reduce\ (+)\ .\ map\ (\hat{}2). \end{aligned} \tag{3.14}$$

sumsquares first squares ($\hat{}2$) all elements of the input stream, and then sums them up ($+$).

Finally, Brook kernels can have variable-length output streams using the `vout` specifier:

Program 3.6 returns the differences between adjacent stream elements for those elements which differ by more than t. Again, we can separate the variability from the generation of the output, by applying a *filter* skeleton. *filter* takes a function

Program 3.6 Brook kernel function to return those gradients of which the value is greater than a certain threshold

```
void kernel gradthres(float a<>[2], in threshold, vout[1] float b<>)
{
  b = abs(a[0]-a[1]);

  if (b > threshold)
    push(b);
}
```

that determines for a stream element whether it should be kept or not, and returns a function that keeps only those stream elements which satisfy the criterion:

$$\begin{aligned} &\mathit{filter} && :: (a \to \mathit{Bool}) \to ([a] \to [a]) \\ \\ &\mathit{diff} && :: [a] \to a \\ &\mathit{diff}\,[x, y] && = \mathit{abs}\,(x - y) \\ \\ &\mathit{gradthres} && :: t \to ([a] \to [a]) \\ &\mathit{gradthres}\,t && = \mathit{filter}\,(> t)\ .\ \mathit{map}\ \mathit{diff}\ .\ \mathit{stencil}\,2. \end{aligned} \tag{3.15}$$

First, we generate the stencil stream in order to have access to the neighborhood. Next, we calculate the difference between the adjacent stream elements. Finally, we filter out the differences that are smaller than the threshold.

We can view the Brook kernel language as a skeleton that places few restrictions on its input function: stencils, reductions, and variable-length outputs are all supported. This limits the exploitable parallelism, and means that general Brook kernels can only be compiled to architectures which support all those features.

We observe that different aspects of Brook kernels can be separately implemented using different skeletons. Each of these skeletons places different restrictions on its input function, and has different execution characteristics. If the user chooses the most restrictive skeleton for each (sub)kernel, the largest amount of parallelism can be extracted from the application, and the least amount of requirements are placed on the hardware.

3.4.1 Generalizing Stream Kernels

map, *reduce* and *filter* are just a few of the possible skeletons; looking at chapter 2, recursive neighborhood operations and stack operations cannot be captured by them. We can extend the applicability of stream programming by allowing different types of skeletons to be used. These extensions come in three different modes:

1. They may *generalize* an existing skeleton, giving the kernel more freedom but limiting the portability and parallelism. The progression from pixel operation → neighborhood operation → recursive neighborhood operation

is an example of this mode. A global operation, allowing random access to both the entire input and the entire output, is the final stage.

2. They may *specialize* a more general skeleton in order to allow execution on hardware with specific limitations. For example, the convolution function in the FPGA platform depicted in figure 2.4 only supports linear neighborhood operations. We can create a skeleton that only allows the specification of kernel coefficients, and implements the convolution itself.

3. They may be specialized for the natural specification of a specific class of operations. These operations can be specified using another skeleton or combination of skeletons, but it would be cumbersome to do so. An example is a propagation implemented by iterative (recursive) neighborhood operations. This may be implemented in a loop, testing for idempotence every iteration, but an integrated idempotence test would be both simpler and more efficient.

Since we can never be sure which hardware is going to be used, and which operations are going to be executed on it, it is important that we do not restrict the user to a certain set of predefined skeletons. Rather, we allow a specialized programmer, if not the user himself, to add his own skeletons, to support new hardware or new types of operations.

3.5 Discussion

In this chapter, we have explained how architecture-independence can be achieved by using algorithm-specific languages and stream programming. Algorithm-specific languages were introduced as a generalization of algorithmic skeletons, allowing the user to write stream kernels in an architecture-independent way by placing different restrictions (and therefore hardware requirements and parallelization opportunities) on different kernels. This means we can efficiently map the stream graph to a heterogeneous system.

Other systems have been used to write programs in terms of processing and communication, such as *TTL* [128] (based on Kahn process networks). These systems generally view an application as a set of processes that communicate over some channels, and place different restrictions on the allowable interactions. They neither specify how the individual processes are to be written, nor do they allow for the dynamic, data-dependent reconfiguration of the channels. Rather, reconfigurations are seen as special events that happen infrequently [99].

Algorithmic skeletons have previously been applied in the field of image processing by [119]. However, this work was limited to the *map* skeleton and homogeneous systems. It also suffered from communication inefficiencies, because data was gathered to a control node after every processing step. Another approach, based on an abstract parallel image processing machine, eliminates this overhead but does not support task parallelism [114].

Algorithm-specific languages can be seen as a continuation of the trend towards *domain-specific languages* (DSLs), where each algorithm class is its own domain.

While it is not uncommon for more than one DSL to be used in a single program [23], using a different one for each class of algorithms is.

Another related approach is that of *active libraries* [44] and *interface compilation* [48], where a library (which can be seen as a domain-specific interface) takes an active role in compilation. C++ template meta-programming [131] can be used to write such libraries. Unlike stream kernels written in an ASL, however, they do not support target architectures which cannot implement the full host language, since only a single program is generated.

We will continue in chapter 4 by describing how we implement skeletons and algorithm-specific languages in an extensible way using a technique we call pseudo-dynamic meta-programming. Chapter 5 then details our implementation of stream programming, and chapter 6 shows how we use the resulting architecture-independent program to automatically find an appropriate heterogeneous multiprocessor architecture.

Chapter 4

Implementing skeletons using meta-programming

The task of a skeleton can be described as the source-to-source translation of a kernel, written in an algorithm-specific language with an algorithm-specific interface, into an operation, written in a target-specific language and interfacing with the target's run-time system (section 3.2.3). The skeleton's algorithm-specific interface generally specifies a computation on one element, while the resulting operation has to perform this repeatedly to process an entire data structure, such as an image.

This chapter deals with how skeletons can transform a kernel into an operation, and how kernels can be defined such that the run-time system has the information that it needs. In section 4.1 we will first set the stage by defining a skeleton's functional requirements, that is, what its input and desired output are. Based on this information, section 4.2 will review the field of meta-programming, and distill the most convenient way of performing the necessary transformations. In section 4.3 we describe our approach, which we call *pseudo-dynamic meta-programming* and which is based on the concepts of rewriting (section 4.4) and partial evaluation (section 4.5). Section 4.6 discusses the merging of skeletons to improve efficiency, and section 4.7 presents a performance evaluation of the skeleton approach. Finally, we discuss our results in section 4.8.

4.1 Functional requirements

The input to a skeleton is a kernel definition. As each kernel may be transformed by a different skeleton, they should specify which skeleton to use. Continuing in the terminology of section 3.2.3, this limits the subset of `S` that the kernel can utilize. The output is a function definition, using a target processor's run-time system to perform the operation. A different skeleton implementation will have to be written for each target language or run-time system.

4.1.1 Defining kernels

A kernel definition should contain the information that the run-time system needs to set up the correct connections (see section 3.3.3). This includes which parameters are streams and which are scalars, which parameters are inputs and which are outputs. It should also define the skeleton that is going to be called, and the information the skeleton needs to perform the transformation, such as the data types of the parameters.

Our kernel definitions resemble Brook [25] kernels (such as program 3.3), with the major exception that they specify a skeleton. Program 4.1 shows an example of an erosion kernel.

Program 4.1 Definition of an erosion kernel, using the `NeighborhoodToPixelOp` skeleton.

```
NeighborhoodToPixelOp()
erosion(in stream unsigned char src[-1..1][-1..1],
        out stream unsigned char *dst)
{
  int y, x;
  unsigned char p = 255;

  for (y = -1; y <= 1; y++)
    for (x = -1; x <= 1; x++)
      if (src[y][x] < p)
        p = src[y][x];

  dst = p; // output smallest value in neighborhood
}
```

This kernel should be transformed by an implementation of the `NeighborhoodToPixelOp` skeleton, and has 1 stream input and 1 stream output, both `unsigned chars`. The stream input accesses a 3x3 neighborhood stencil.

The kernel code itself closely resembles C. It references the stencil with a *relative* index. This relative indexing is the algorithm-specific interface of the `NeighborhoodToPixelOp` skeleton, which has to implement it.

4.1.2 Generating operations

There will be multiple implementations of `NeighborhoodToPixelOp`, each translating the kernel into a different (parallel) C dialect. Program 4.2 shows a possible output for the TriMedia processor, while program 4.3 is targeted towards the IMAP processor. Both programs read from and write to FIFO buffers, in accordance with the run-time interface. The TriMedia program then loops over the pixels, while the 1DC program processes them in parallel.

Consider the transformations that have to be made to the kernel definition in order to produce the output. The most obvious is the addition of a great deal of buffer interaction and looping code, identified in section 3.2.2 as the structure of a computation. This is purely *generated* code, dependent only on the parameters

Program 4.2 TriMedia version of the erosion operation, generated from program 4.1. Lead-in and lead-out (for vertical border handling) are omitted for clarity.

```
void NeighborhoodToPixelOp_erosion(buffer *srcbuf, buffer *dstbuf)
{
  int stride;
  unsigned char *restrict src[3], *restrict dst;

  stride = bufferGetStride(srcbuf);
  while (bufferPeek(srcbuf, &src[0], stride))
  {
    int i;

    bufferAllocate(dstbuf, &dst, stride);

#pragma TCS unroll=16
    // Process stride elements
    for (i=0; i < stride; i++)
    {
      int y, x;
      unsigned char p = 255;

      for (y=-1; y <= 1; y++)
        for (x=-1; x <= 1; x++)
          if (src[y+1][i+x+1] < p)
            p = src[y+1][i+x+1];

      dst[i] = p;
    }

    bufferReleasePeeked(srcbuf, stride);
    bufferReleaseAllocated(dstbuf, stride);

    // Shift vertical neighborhood
    src[2] = src[1];
    src[1] = src[0];
  }
}
```

Program 4.3 1DC version of the erosion operation, generated from program 4.1.

```
void NeighborhoodToPixelOp_erosion(buffer *srcbuf, buffer *dstbuf)
{
  int stride;
  sep unsigned char *src[3], *dst;

  while (bufferPeek(srcbuf, &src[0]))
  {
    int i;

    bufferAllocate(dstbuf, &dst);

    {
      int y, x;
      sep unsigned char p = 255;

      for (y=-1; y <= 1; y++)
        for (x=-1; x <= 1; x++)
          if (src[y+1]:<x < p)
            p = src[y+1]<:x;

      *dst = p;
    }

    bufferReleasePeeked(srcbuf);
    bufferReleaseAllocated(dstbuf);

    src[2] = src[1];
    src[1] = src[0];
  }
}
```

of the kernel. For example, the number of lines that need to be shifted depends on the vertical neighborhood size.

It is instructive to note that the TriMedia code includes the `restrict` qualifier (signifying no aliasing), and an explicit loop unrolling `pragma`. These are required for the compiler to generate efficient code. Thus, even while TriMedia is considered to be ANSI C programmable, it requires special attention to efficiently exploit all its features. Other "C" programmable processors will generally require different measures.

The second part of the translation consists of *rewriting* the kernel to make use of the provided structure, and possibly translating it into the output C dialect. For the TriMedia version, this is limited to changing the mode of array indexing, but the 1DC code requires some more work. First, the horizontal array indexing has been replaced by the 1DC *shift* operator, and second, the local `p` variable has been upgraded to a `separate` type, because its value depends on `src` and is therefore different for each processing element.

Summarizing, we can distinguish three distinct transformations, presented in order of increasing complexity:

- Code generation, for the specification of structure
- Pattern substitution, to translate the algorithm-specific interface to the interface provided by the run-time system
- Program analysis and rewriting, to translate the kernel into the target C dialect

Since our library of skeletons must be extendable, a skeleton implementation should be able to easily specify all three kinds of transformations. This casts the skeleton into the role of a *meta-program*: a program that manipulates other programs. We will introduce the field of meta-programming in section 4.2, and present our own meta-programming language and tool, called PEPCI, in section 4.3.

4.1.3 Compilation flow

Recall the kernel definition in program 4.1. A stream program consists of a number of those kernel definitions, plus glue code to connect them together. Each kernel definition will have to be separately processed by a skeleton. The job of our compiler (called the SmartCam compiler), is to extract the kernel definitions and call their skeletons, gathering all operations for each target in a coprocessor program. Calling a skeleton means handing the kernel code and associated information to our PEPCI meta-programming tool, yielding an operation.

After all kernel definitions have been processed, the remaining glue code is plain C, which runs on a control processor to coordinate the execution. Figure 4.1 illustrates this basic compilation flow.

4.2 Meta-programming

Meta-programming [117, 42, 43] concerns programs that operate on other programs. This is a very large field, encompassing many different approaches. We will there-

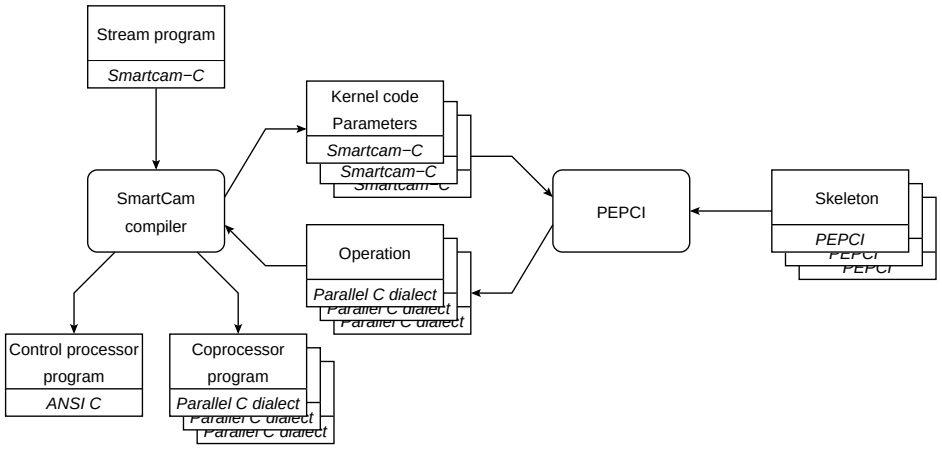


Figure 4.1: Compilation flow of the SMARTCAM compiler.

fore provide a brief overview and distill the most appropriate way of implementing our skeletons.

We can distinguish three classes of meta-programs: *program generators*, which generate a program based on some input parameters, possibly including an input program as part of the output, *program analyzers*, which analyze an input program and provide information about it, and *program transformers*, which analyze an input program transforming it into an output program.

Depending on the situation, meta-programs can deal with three distinct languages. The *source language*, in which the input (or source) program is written, the *object language*, in which the output (or object) program is written, and the *meta language*, in which the meta-program itself is written. For program generators, and many program transformers as well, the source language is the same as the object language. Reflective languages, which use (run-time) meta-programming to change themselves, generally use only one language.

Semantics-preserving program transformers with different source and object languages are called *translators*. The most well-known program translators are *compilers*, which translate a program written in a high-level language into a functionally equivalent low-level assembly language.

This section is based partly on Sheard's taxonomy of meta-programming systems [117], but we focus on systems which support C meta-programming.

4.2.1 Representation

The most important aspect of a meta-programming system is the way in which the object program is represented in the meta-program. We will therefore first discuss the three most important representations.

Opaque

This means that the representation cannot be deconstructed by the meta-program (it cannot be divided into parts). This representation is limited to program generators. Examples are the various web scripting languages (ASP [136], JSP [111], PHP [2]), in which the meta-program is separated from the object program (an HTML web page) using `<% %>` *code render blocks*. The object program code is simply written out. Higher-order functions [20] also use an opaque representation, as they cannot access the input function's representation, only call it. Unless specific measures are taken, C++ template meta-programming [131, 84] falls into this category as well.

String based

Object programs are represented as simple text strings, and can be manipulated using standard string manipulation functions. While in principle this allows arbitrary manipulation and analysis, substitutions are made on a lexical instead of a syntactic basis, and are therefore verbose and error-prone. Examples of this are the M4 [77] macro language and `sed` scripts.

Abstract syntax trees

Abstract syntax tree (AST) describe how a program (fragment) is parsed, thus providing a syntactic basis for manipulations (see figure 4.2). For many program transformations this is the most natural form, and therefore almost all dedicated meta-programming systems support it. *Expression templates* [130] are a way of accessing the AST using C++ template meta-programming, by constructing a type hierarchy. The Sh language [96] uses C++'s *operator overloading* to construct an object hierarchy instead.

Unfortunately, AST manipulation requires knowledge of the language's grammar on the side of the meta-programmer, and this is a great disadvantage. To avoid this, many systems allow programmers to *quasi-quote* [106] object language code. The abstract syntax trees can thus be constructed implicitly using a *concrete object syntax* [134]. Assuming ‘ is the quasi-quotation operator, the concrete object syntax for the syntax tree in figure 4.2 is simply `‘pi = 4 * atan(1);‘`.

4.2.2 Levels and Staging

A program analyzer or transformer is, by definition, a multi-level language. This means that variables may hold program code. A language is two-level if the program code held in variables may not itself contain variables that hold code values. Conversely, an N-level language allows variables at all levels to hold program code. Code generators do not require variables to hold code values. They do require code *constants*, however, be they strings, templates, code render blocks or abstract syntax trees.

Stages arise when the execution of a program is distributed over different points in time. The most common case is when a program is first compiled and then

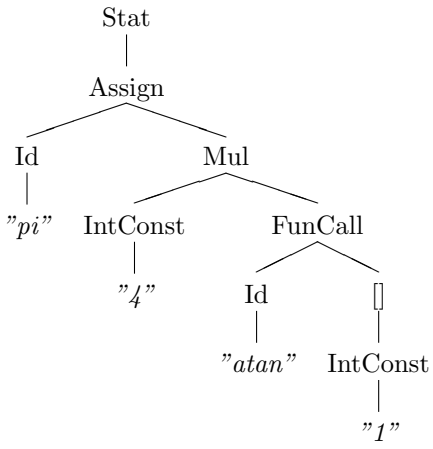


Figure 4.2: Abstract syntax tree representation of the statement "`pi = 4 * atan(1);`".

executed: this is two-stage execution. If the program has to be generated first, there are three stages, etc.

Often, staging is done manually. This means that the programmer has explicit control over which computations occur at which times. For example, C++ templates will always be instantiated at compile-time. In other cases this is not so clear, as compilers will often apply optimizations such as constant folding and propagation to speed up execution.

In the field of *partial evaluation* [73], which aims to carry these optimizations through as far as possible, this automatic staging is called *binding time analysis*. First, the inputs to a program or function are labeled as either *static* or *dynamic*. By program analysis or abstract interpretation, a partial evaluation tool propagates this information in order to discover which program fragments depend on dynamic values, and which depend only on static values. The static program fragments are then executed, while the dynamic parts are output as the *residual program*.

The obvious advantage of automatic staging is that it is not necessary to manually switch between the object and meta level. In situations where switching is frequent, this can be very convenient. Examples of systems which implement partial evaluation of C programs are Tempo [39] and C-Mix [10, 90].

4.2.3 Static versus dynamic

A meta-programming system can be either *static* or *dynamic*. Static meta-programming systems (such as SUIF [4] and ANTLR [102]) run at compile-time and cannot execute the programs they generate. Dynamic systems can analyze, create and execute new programs at run-time.

Dynamic meta-programming has two main advantages. First, the program is able to adapt to dynamic changes in the program's environment. For example,

it may create special, instantiated functions for oft-occurring values in the input. Second, executing created programs instead of writing them to disk means working in a different mind set. Instead of writing a program which generates a program to process some input, the user just writes a program to process the input directly (perhaps analyzing and creating programs on the way for faster execution). 'C (TickC, [105]) is a compiler and run-time system which provides dynamic meta-programming for C.

A distinct disadvantage of dynamic meta-programming is the time it takes to compile or interpret the generated programs at run-time.

4.2.4 Homogeneity

It has been noted that meta-programming systems can deal with up to three different languages. We call meta-programming systems that use only one language *homogeneous*. Homogeneous meta-programming systems have many advantages:

- The user has to learn only a single language.
- The system can be N-stage, where a meta-program itself can be subject to meta-programming.
- Interaction between dynamically generated programs and the meta-program is straightforward.

Since C does not natively support meta-programming in a convenient way, it is necessary to extend the language in order to support homogeneous meta-programming. A disadvantage is therefore that the basic imperative, control-flow-driven design of C (and C++) is not ideal for meta-programming.

4.2.5 Typing

Type systems are an important tool for avoiding programmer errors. An untyped object program representation may result in type errors when the object program is compiled by a downstream compiler. Worse, the type errors may be suppressed by explicit type conversions, resulting in an erroneous executable.

Meta-programming type systems range from very strict (such as MetaML [122]'s strong static staged typing) to nonexistent (such as string manipulation). We can distinguish four points at which type errors can be caught. First, as in MetaML, they may be caught during compilation of the meta-program. This provides the most direct feedback. Second, they may also be caught during *execution* of the meta-program, such as in SUIF, where typed abstract syntax trees are manipulated. A third point is during compilation of the object program. This is where type errors from C++ template meta-programs are caught, and this can result in cryptic error messages because it may be difficult to analyze which part of the meta-program is responsible for the error. Finally, the type error may be caught during execution of the object program. This may happen if the object program language is dynamically typed. Of course, the error might not be caught at all and result in erroneous outputs or memory corruption.

4.2.6 Discussion

Reviewing the previous sections we can distill, from a usability perspective, the most convenient features for a meta-programming system for C skeleton instantiation:

- The object program representation should be easy to construct, deconstruct and manipulate. This means an AST representation that is accessible using concrete object syntax.
- The program should be automatically staged. In a multi-level language, this requires support for dynamic meta-programming.
- The language should be homogeneous.
- Type errors should be caught at the meta-program level.

Our source and object languages are C and C derivatives. This creates a contradiction, since they do not support meta-programming. We therefore relax our desire for the system to be homogeneous, instead stating that the metalanguage should be similar to the object language while still naturally supporting meta-programming.

Dynamic meta-programming, however, requires the metalanguage to run on the target, and this is not possible. Thus, we require all meta-programming extensions to have static binding times, so that after automatic staging the residual program contains only object language syntax. We call this *psuedo-dynamic meta-programming*, and will detail our implementation in the next section.

4.3 A meta-programming language for skeleton instantiation

Based on the design goals set out in the previous section, we have designed a new meta-programming language for skeleton instantiation. This language, called PEPCI, targets a *virtual* platform that supports dynamic meta-programming. This platform is virtual, because none of our downstream compilers support dynamic meta-programming. Rather, we require all meta-programming to depend on properties of the kernel, and not on the values supplied to it at run-time. Since the kernel is available at compile time, we can use *partial evaluation* to evaluate all meta-programming constructs at compile time, resulting in a single-stage program. Using dynamic meta-programming as well as partial evaluation satisfies two of our design goals: we do not need to switch between meta-level and object level for code generation, and the generated program is guaranteed to be type-safe (as the partial evaluator would complain otherwise).

Since our target languages are C derivatives, PEPCI is also based on C, thus satisfying similarity. Multiple object languages are supported using the modular SDF2 Syntax Definition Formalism [72, 132]. Adding a new object language thus consists of writing a grammar for it, and adding the appropriate handlers to the partial evaluator. This last aspect is simplified by the fact that all object language extensions, by definition, have dynamic binding times.

Appendix B.2 contains the syntax definition of the additions PEPCI makes to the ISO/ANSI C99 standard [139], as well as the major restrictions. The syntax is inspired by that of ‘C [105].

4.3.1 The `code` data type

PEPCI adds a new data type, `code`, to represent object program fragments. None of the usual C operators are defined on this type, except assignment (values of this type may therefore be passed to functions as usual). An object code fragment may be constructed using the ‘ (backtick) quasi-quotation operator. These code fragments are dynamically scoped; they simply represent untyped abstract syntax trees. Interaction with these fragments can be made more structured by embedding them in *lambda expressions*, structures that also provide the names and data types of unbound variables in the code.

A consequence of dynamic scoping is that some code fragments will be ambiguous. Consider the fragment `‘a * b;‘`. The correct parsing of this code is *context-sensitive*. If `a` is a type, it is a declaration of a pointer. If, on the other hand, `a` is a variable, it is simply a multiplication after which the result is discarded. Because the fragments may need to be manipulated before their scope is clear, we restrict them to a *context-free* subset of the language.

In fact, all of PEPCI is context-free. We avoid ambiguities in C using a few simple *disambiguation rules*:

- Prefer identifiers above type names. `‘a‘` is an identifier.
- Prefer declarations above statements. Thus, `‘a * b‘` is parsed as an expression, while `‘a * b;‘` is parsed as a declaration.
- Prefer function calls above typecasts. `‘(a)(b)‘` is a function call. Typecasts of this nature must be specified using a `cast` library function.
- Whether the parameter of `sizeof` is a type or an expression is determined at run-time.
- Old-style function prototypes are not allowed. `‘foo(a, b, c);‘` is a function call, and not a function prototype with implicit types.

4.3.2 Execution

Since PEPCI is a dynamic meta-programming language, we allow the execution of code fragments. This is done using the *code dereference operator* “`@`”. This evaluates the contents of a code variable in much the same way that a pointer dereference accesses the contents of a pointer. A code dereference may occur in most places where an identifier or type name is expected.

Since code fragments represent untyped parse trees, dereferencing them may result in *specialization-time type errors*. For example, in program 4.4, the last line will generate a type error if “`f`” is not a function. As all code dereferencing has to have static binding time, all type errors will be caught at specialization time.

Program 4.4 Examples using the code dereference operator "`@`".

```
code typeid = `int`, id = `a`, exp = `1 + 2`, fun = `f`;

@typeid @id; /* Declare a of type int */

@id = @exp; /* Assign 3 to a */

(@fun)(@id); /* Call f with parameter 3 */
```

4.3.3 Splicing

Until now, we have dealt with static code fragments. However, the point of code generation is to create code programmatically. Templates (code fragments with holes, similar to string format specifiers) have been described as an appropriate way of doing this, and are supported by PEPCI using the *splice operator* "`$`". This operator may only occur within a code fragment, and results in splicing the value of its meta-level variable argument into the code.

Program 4.5 Generating code using templates and the "`$`" splice operator.

```
1  code power(code c, int n)
2  {
3    if (n)
4      return `$@c * $@power(c, n-1)`;
5    else
6      return `1`;
7  }

8  int x = 4;
9  printf("x^3 = %d\n", @power(`x`, 3));
10 // power(`x`, 3) = `x * $@power(`x`, 2)` = ...  = `x*x*x*1`
```

Note the use of `$@` in line 4 of program 4.5. We are thus splicing the dereferenced value of `c` in the code. Otherwise, `` `x` `` would be inserted instead of `x`. Because of scoping concerns, only statically bound variables may be spliced.

4.3.4 Lambda expressions

Untyped abstract syntax trees and dynamic scoping are dangerous concepts. It is therefore encouraged to encapsulate code fragments in a *lambda expression* in order to capture unbound variables.

Lambda expressions are supported using the library functions `lambda` and `apply`. Program 4.6 shows how a specific exponential (3, in this case) can be encapsulated and later applied using a lambda expression. The lambda expression captures the variable `x`, so that the code no longer depends on the *name* `x`. If all unbound variables are captured by the lambda expression, the scope of the code is no longer dynamic.

Program 4.6 Encapsulating code fragments using lambda expressions. Assume `power` is defined as in program 4.5.

```
lambda_t powerlambda(int n)
{
  return lambda(power(‘x‘, n), ‘int‘, ‘x‘);
}

lambda_t power3 = powerlambda(3);

printf("4^3 = %d\n", @apply(power3, 4));
```

4.4 Rewriting

The concepts described above do not allow the analysis and deconstruction of code fragments: code is essentially *opaque*. Rather than adding more language constructs, PEPCI provides a number of library routines to deal with code deconstruction and rewriting. These provide code comparison, pattern matching and replacement, type inspection, etc.

Another problem with the presented metalanguage is that not all code fragments can be constructed using just concrete object syntax. For example, it is impossible to construct new identifier names, or a function call with a number of parameters that is unknown in advance. PEPCI therefore has library routines that provide these functionalities for the most common cases.

However, we recognize that for more involved transformations, a C-style library is not the most suitable solution. And, of course, the question arises as to how the library routines themselves should be implemented. In accordance with our statement that the metalanguage should be natural for program transformation, we have chosen to leverage a language that was specifically designed for it: Stratego [133]. Although Stratego breaks the requirement that the metalanguage and object language should be similar, it allows the simple specification of a wide range of transformations in both concrete and abstract syntax.

There is thus a clean break between simple rewritings that can be expressed using concrete syntax (these are accessible using library routines) and more complex transformations that must be specified using abstract syntax (these are done by calling Stratego). A skeleton writer only needs to learn Stratego if he needs such complex transformations. To avoid too large a gap, PEPCI *embeds* Stratego, allowing the specification and transparent calling of rewriting strategies.

4.4.1 The Stratego term rewriting language

Stratego [133] is a language for program transformation based on the paradigm of strategic term rewriting. It makes a distinction between *rewrite rules*, and *rewriting strategies* that repeatedly apply those rules. A rewrite rule consists of a pattern that has to be matched, and a result term that will replace the pattern. A pattern may contain free meta-variables that will be unified upon matching. These can then be used in the result term.

Program 4.7 Stratego code to apply a simple math identity.

Simplify:
Add(e, IntConst("0")) → e
Simplify:
Add(IntConst("0"), e) → e

In program 4.7, `Simplify` is a rewriting rule that can match two patterns, both factoring out addition with zero. The patterns are defined using abstract syntax, `e` being a metavariable. Patterns may also be defined using concrete syntax, such as in program 4.8. Here, `~:` is an anti-quotation operator, meaning that `e` has to be interpreted as a meta-level variable instead of an object program variable.

Program 4.8 Stratego code to apply a simple math identity using concrete syntax.

Simplify:
|[`1 * ~:`e]| → e
Simplify:
|[`~:`e `* 1`]| → e

A rewriting strategy specifies which rules to apply, and in which order. This is done by combining simple strategies in a number of ways. One of these strategies is `all(s)`, which applies a strategy `s` to all direct sub terms of an AST node. Using `all`, we can define several common traversals:

Program 4.9 Defining traversals and using them to perform a transformation.

bottomup(s) = **all**(**bottomup**(s)); s
topdown(s) = s; **all**(**topdown**(s))
alltd(s) = s <+ **all**(**alltd**(s))

simplify = **bottomup**(**try**(**Simplify**))

The `;` and `<+` *strategy combinators* denote sequential composition and deterministic choice, respectively. `s ; t` will first apply `s`, and then apply `t` to the result. `s <+ t` will first try to apply `s`, and only applies `t` if `s` *fails*, that is, no pattern can be matched.

`bottomup(s)` will first recursively apply itself to all of a node's sub terms, thus starting at the leaf nodes. Then, it applies `s` to result. `topdown`, on the other hand, first applies `s`, and recurses afterwards. Finally, `alltd` is similar to `topdown`, but only recurses if `s` fails. `s` is therefore applied along a *frontier* of a term.

We have chosen the `bottomup` strategy for `Simplify`. It will try to apply the `Simplify` rewrite rule to all nodes of the AST, starting with the leaf nodes. `try(s)` is defined as `s <+ id`, where `id` is the identity strategy.

This way of programming contrasts with implicit rewriting strategies (such as `innermost`, which exhaustively applies a strategy to a term, starting with the innermost sub terms) [21, 22], and defining the rewriting strategy within the rules

themselves (using explicit recursion). Custom rewriting strategies allow more opportunities for reuse by separating the structure of the rewriting from the specifics in much the same way a skeleton separates the structure of a computation from the computation itself.

4.4.2 Embedding Stratego

PEPCI *embeds* the Stratego language. By this we mean that Stratego syntax is allowed in certain parts of a PEPCI program, and that the program transformations defined by those Stratego fragments can be transparently called by the user. A Stratego transformation is written as a normal function definition, but containing Stratego clauses instead of C declarations and statements.

Program 4.10 Embedded Stratego function to strip a pointer from a pointer type. The type is passed as the current term of the `main` strategy, which tries to apply the `StripPointer` rewrite rule to it.

```
code strippointer(code type)
{
  strategies
    main =
      try(StripPointer)

  rules
    StripPointer:
      TypeName(spec, Some(Pointer([p1,p2|pointers], bounds))) →
      TypeName(spec, Some(Pointer([p2|pointers], bounds)))
    StripPointer:
      TypeName(spec, Some(Pointer([pointer], bounds))) →
      TypeName(spec, bounds)
}
```

Program 4.10 shows a strategy that strips a pointer from a pointer type. The function's arguments are passed as the subject term of the strategy (if there is more than one argument, it is passed as a *tuple*). The strategy tries to match a `TypeName` with either one pointer or a list of more than one pointer. In both cases, the first pointer is stripped off the type.

Additionally, the function's arguments are available as pre-bound variables. The `main` strategy may therefore be rewritten as in program 4.11. `<s>(t)` means the application of strategy `s` to term `t`.

These kinds of embeddings are facilitated by using the modular SDF2 Syntax Definition Formalism. An SDF2 specification consists of a number of modules which together define a context-free grammar. We can combine two languages by importing their respective modules, and writing extra productions which realize the embedding. Program 4.12 shows the essence of how PEPCI embeds Stratego.

This module imports the PEPCI and Stratego grammar specifications, and provides one production rule, stating that a function declaration followed by a

Program 4.11 Using pre-bound variables in an embedded Stratego function, thereby making the application of `StripPointer` to `type` explicit.

```
stratego code strippointer(code type)
{
  strategies
    main =
      <try(StripPointer)>(type)
  ...
}
```

Program 4.12 Embedding Stratego using SDF2. A new grammatical production for the `FunDef` sort allows the use of Stratego syntax in a PEPCI program.

```
module P-Strategies
  hiddens
    imports P StrategoRenamed

  exports
    sorts FunDef
    context-free syntax
      FunDefDecl "{" StrategoDecl* "}" → FunDef {cons("StrategoDef")}
```

number of Stratego declarations enclosed in braces may be parsed as a function definition, and is called a `StrategoDef` in the resulting abstract syntax tree. A full grammar for the PEPCI language can be found in appendix B.

4.4.3 Interpreting embedded Stratego

In order to actually perform the transformations defined in PEPCI, we need to execute the embedded stratego fragments. Figure 4.3 illustrates this.

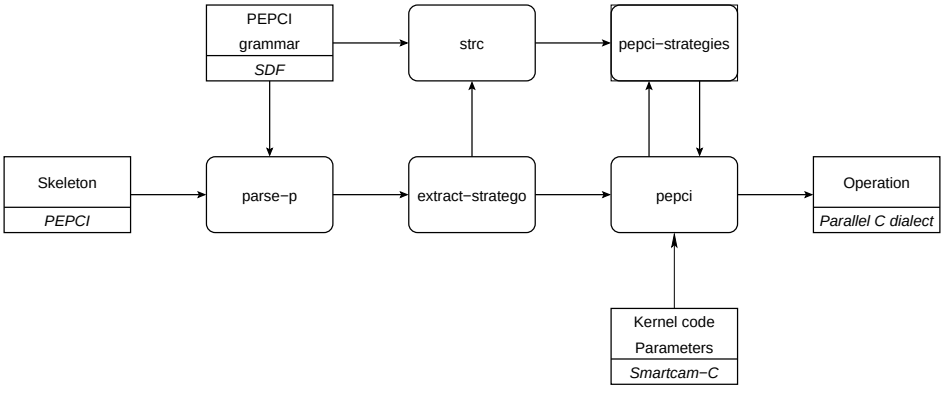


Figure 4.3: Information flow of the PEPCI interpreter.

At the start of interpretation, the program is parsed using the combined grammar; the result is an abstract syntax tree in ATerm [126] format. The Stratego definitions are extracted from the AST, pretty-printed, and gathered in a combined Stratego source file. This file is then compiled using the Stratego compiler, `strc`, resulting in an executable transformation tool which implements all of the inline strategies.

The program code without Stratego definitions is passed to the interpreter. When one of the strategies is called during interpretation, the subject terms are converted to ATerm format, and the transformation tool is called, selecting the appropriate strategy. The result is parsed into PEPCI's native format and processed further. A failing strategy will cause the interpreter to terminate.

All values are passed by their constant representation. For example, an integer value of `1` would be passed as if it were `‘1‘`, with an AST representation of `IntConst("1")`. Similarly, arrays are passed as if specified in a declaration initializer such as `‘{1, 1, 2, 3, 5}‘`. Consequently, neither addresses nor dynamic values may be passed to a Stratego function.

4.5 Partial evaluation

Partial evaluation, also called *program specialization*, is mainly used to speed up the execution of a program by computing as many values at compile-time as possible. If the values of certain arguments to a function or program are known beforehand, the program can be *specialized* with regard to those values, hopefully resulting in a faster program.

Theorem 4.1 (Program Specialization) *If p_S is a two-input program written in language S, then*

$$out = [\![p_S]\!]_S(in1, in2), \tag{4.1}$$

where $[\![\cdot]\!]_L$ is the program meaning function *for a program written in language L. If $pe(S \to T)$ is a partial evaluator for S with target language T (itself written in some unspecified language), and in1 is static, then*

$$\begin{aligned} p_T^{in1} &= [\![pe(S \to T)]\!](p_S, in1) \\ out &= [\![p_T^{in1}]\!]_T(in2) \end{aligned} \tag{4.2}$$

is the two-stage execution *of p, where p_T^{in1} is the* specialization *of p with regard to in1.*

Dynamic meta-programming, such as supported by Lisp and ‘C, is often associated with the interpretative or code generation overhead. However, if the created code depends only on values that are available at compile time, the interpretation may be removed by the *first Futamura projection* [56].

Theorem 4.2 (First Futamura Projection) *Consider $int(S)_L$, an interpreter for S written in L, that is, for all valid programs p_S and inputs in*

$$[\![p_S]\!]_S(in) = [\![int(S)_L]\!]_L(p_S, in). \tag{4.3}$$

If such an interpreter is specialized with regard to a program, we get, by equation 4.2

$$\begin{aligned} int(\mathcal{S})_T^{p_S} &= [\![pe(L \to T)]\!](int(\mathcal{S})_L, p_S) \\ out &= [\![int(\mathcal{S})_T^{p_S}]\!]_T(in). \end{aligned} \tag{4.4}$$

Combining equations 4.3 and 4.4, we get

$$[\![p_S]\!]_S = [\![[\![pe(L \to T)]\!](int(\mathcal{S})_L, p_S)]\!]_T, \tag{4.5}$$

that is, interpreter specialization amounts to translation.

Although theorem 4.2 does not guarantee speedup, the translated program can be expected to be faster than interpreting the source, since most interpretation overhead is dependent on $p_{\mathtt{S}}$, and not on *in*. This has been used in the context of meta-programming by [92]. In their case, an object is associated with a meta-interpreter, which interprets the object methods' code upon invocation. These meta-interpreters are specialized with regards to the object code, removing the interpretation overhead.

4.5.1 Language specialization

As described before, PEPCI is extended with multiple object languages. Our L is therefore the union of a core language P and an object language T:

$$\mathtt{L} = \mathtt{P} \cup \mathtt{T}. \tag{4.6}$$

Instead of writing an interpreter for some kernel language S and specializing this to a T-program, a PEPCI program uses term rewriting to translate the kernel into a T-program fragment. Using dynamic meta-programming, the fragment is then executed as part of the skeleton:

$$\begin{aligned} out &= [\![kernel_{\mathtt{S}}]\!]_{\mathtt{S}}(in) \\ &= [\![skeleton_{\mathtt{L}}]\!]_{\mathtt{L}}(kernel_{\mathtt{S}}, in). \end{aligned} \tag{4.7}$$

Our targets do not support dynamic meta-programming, however. This means that we do not have a direct way of determining the meaning of an L-program. Rather, we have constructed a partial evaluator which evaluates static P-expressions, and does not touch other code. If, for a program $p_{\mathtt{L}}$, $\mathtt{P} \cap \bar{\mathtt{T}}$ is static (that is, all P-constructs that are not also T-constructs can be evaluated at compile time),

$$[\![pe(\mathtt{L} \to \mathtt{L})]\!](p_{\mathtt{L}}, in) = p_{\mathtt{T}}^{in}. \tag{4.8}$$

Our L-program is *specialized* to an L-subset, namely T. It is thus a hard requirement that all meta-programming depends only on the kernel. This ensures that only T constructs remain in the specialized program.

We rely on downstream compilers to optimize T programs. The performance impact of not specializing T constructs is therefore limited, but extending the partial evaluator with a new target language becomes much easier. Only the syntax and a limited semantics (for correctly handling P expressions within T constructs) are necessary.

4.5.2 Partial evaluation by interpretation

In section 4.2.2 we described partial evaluation as having separate binding time analysis and evaluation phases. This is called *off-line* partial evaluation, and means that the binding times cannot depend on the actual *value* of the inputs. As our inputs can be programs that are dynamically executed, the partial evaluator would have to be very conservative in determining the binding times.

The strategy we use, called *on-line* partial evaluation, does use the input values to determine the binding times. An online PE can be implemented as an interpreter in which each variable in the symbol table can have a special value, `Dynamic`. If a variable is `Dynamic`, all calculations with that variable will result in a `Dynamic` value as well.

An interpreter can be seen as a function, `reduce`, reducing each expression to a value and each statement or declaration to `void`. During reduction, side effects such as changes to the symbol table or IO can occur. Calculations involving dynamic values need to be *residualized* (output to the next stage as part of the residual program), so in that case our `reduce` function returns an `Expression` representing the calculation. Such an expression is treated as a `Dynamic` value in further calculations, see figure 4.4.

Partial evaluation of assignments is more complicated. If the right hand side is static no code needs to be produced, and if it is dynamic the assignment should be residualized. However, any residual dynamic expressions might require assignments that were static at earlier times to be residualized as well.

Consider program 4.13. If `STATIC` is defined, the residual program will consist only of the last statement (because `printf` is an external function that cannot be evaluated):

printf(`"Fibonacci number %d is %d\n"`, 5, 5);

But if `STATIC` is undefined (and `x` is therefore dynamic) the residual expression `a[x-1]` must be entered into the program as part of the last statement. This means that all previously generated elements of `a` must be residualized too, as well as the declaration of `a`. If `N` equals 5, this leads to program 4.14.

Our partial evaluator enters all declarations and assignments into the residual program, but marks them *tentative*. All locations in the symbol table are annotated with the declaration and most recent assignment that produced the

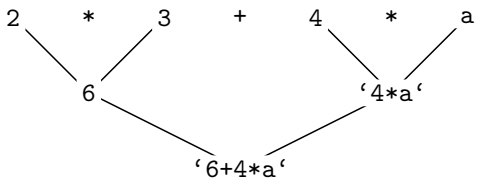


Figure 4.4: Reduction of the expression `‘2*3+4*a‘`. Expressions involving only static values are computed, while expressions with dynamic values are simply returned.

Program 4.13 Program fragment to calculate the Fibonacci sequence. `x` is either static or dynamic, depending on whether `STATIC` is defined.

```
int fib(int *f, int n)
{
  int i;

  for (i=0; i < n; i++)
    if (i < 2)
      f[i] = 1;
    else
      f[i] = f[i-2]+f[i-1];
}

int a[N], x;

fib(a, N);

#ifdef STATIC
  x = 5;
#else
  printf("Enter number (1-%d): ", N);
  sscanf("%d", &x);
#endif
printf("Fibonacci number %d is %d\n", x, a[x-1]);
```

Program 4.14 Residual code of program 4.13, with `STATIC` undefined and `N` = 5.

```
int a[5], x;

a[0] = 1;
a[1] = 1;
a[2] = 2;
a[3] = 3;
a[4] = 5;

printf("Enter number (1-%d): ", 5);
sscanf("%d", &x);
printf("Fibonacci number %d is %d\n", x, a[x-1]);
```

contained value. If the variable occurs anywhere in a residualized expression, the declaration and assignment are unmarked, and will thus be output as part of the residual program.

4.5.3 Flow Control

Flow control constructs which depend on dynamic values introduce the complication that the modifications a dynamic branch makes to the symbol table must be made `Dynamic` after the branch is evaluated, since we can not be sure that the branch is taken. Additionally, both branches of a dynamic `if` statement must be evaluated under the original symbol table. We therefore start the evaluation of each branch with a copy of the symbol table (implemented as copy-on-write to minimize overhead). After completion, all changed symbols are made `Dynamic`.

At this point, a symbol may have up to three assignments: the original one, and one for each branch. When the symbol is used in a dynamic expression (since the symbol itself is dynamic, all expressions including it will by dynamic as well), all three will be residualized. Program 4.15 illustrates this. A symbol is represented as a 3-tuple (*name*, *value*, *assignments*), and removing a symbol from the table is defined as in equation 4.9.

$$ST \setminus x \equiv \{(n, v, a) | (n, v, a) \in ST \wedge n \neq x\} \tag{4.9}$$

Program 4.15 Symbol residualization for dynamic branches. When `y` is used as a dynamic value in line `9`, all three assignments to it are residualized.

1	**int** x;	$ST \leftarrow ST \cup (`x`, \texttt{Dynamic}, \{\})$
2	**int** y;	$ST \leftarrow ST \cup (`y`, \texttt{Dynamic}, \{\})$
3	$y = 0$;	$ST \leftarrow ST \setminus `y` \cup (`y`, 0, \{3\})$
4	**if** (x)	$ST' \leftarrow ST$
5	y=1;	$ST' \leftarrow ST' \setminus `y` \cup (`y`, 1, \{5\})$
6	**else**	$ST'' \leftarrow ST$
7	**if** $(y$==0)	$(`y`, 0, \{3\}) \in ST \Rightarrow$ true
8	y=2;	$ST'' \leftarrow ST'' \setminus `y` \cup (`y`, 2, \{8\})$
		$ST \leftarrow ST \setminus `y` \cup (`y`, \texttt{Dynamic}, \{3, 5, 8\})$
9	**printf**("y=%d\n", y);	residualize lines 3, 5 and 8
		$ST \leftarrow ST \setminus `y` \cup (`y`, \texttt{Dynamic}, \{\})$

Loops are fully unrolled as long as their condition is static. Once they become dynamic (for example, because the loop variable is conditionally updated), their body is evaluated in the same way as the branch of an `if` statement. Dynamic flow control constructs themselves (such as the `if .. else` in lines 4 and 6) are always residualized. Again, our primary goal is not to create an optimal residual program (instead depending on downstream compilers to do the optimization), but to specialize the language.

4.5.4 Dealing with pointers

Pointers are often said to be the worst enemy of C optimization, because aliasing (where a memory location can be accessed through different names) hampers many analyses. In the case of partial evaluation of the full C language, assignment to a dereferenced dynamic pointer must make all values in the symbol table dynamic. It is therefore imperative that pointers are not made dynamic unnecessarily.

We maintain a list of possible addresses for each pointer, in much the same way that a symbol can have multiple assignments. After a dynamic branch, the address list is merged. If such a pointer is written to later on, all symbols reachable through the pointer are made dynamic. A pointer may be assigned to the address of a dynamically indexed array. In that case, the whole array becomes dynamic.

Special care has to be taken with residualized function calls. Since we cannot assume any behavior on part of the called function, the values reachable through any addresses passed to such functions should be made dynamic, and the possible address list of pointer values therein should be extended with those addresses.

This last aspect is currently not implemented in our prototype. We also do not allow pointer arithmetic, and expect dynamic indexing to remain within the memory block allocated for a single array. Furthermore, typecasting pointer values is also not allowed.

4.5.5 Foreign functions

Most function calls to external libraries cannot be specialized even if their arguments are all static, because they may have side effects such as I/O or the writing of global variables. However, some targets do not support the floating point arithmetic necessary to compute some mathematical functions, and we would like to specialize such tasks as computing a Gaussian kernel of a specific standard deviation. We therefore make an exception for (mathematical) pure functions such as `exp` and `sqrt`.

Such a function must be declared `extern`, and will be called at specialization time if all its arguments are static. It will be located in any number of dynamic link libraries provided to the partial evaluator, and called through the `ffcall` [65] foreign function call libraries.

An external function with static arguments that is not embedded in a dynamic control flow statement is allowed to have side effects. This is used by our skeletons to provide information about the kernel and the resulting operation through an interface to write XML files. The information is used by the run-time system and contains, among others, the amount of buffer space the operation requires for its stream arguments.

4.6 Skeleton merging

Until now, we have discussed the implementation of single (data parallel) skeleton calls. In the skeleton literature, such skeletons can be incorporated into task parallel skeletons such as farms and pipelines, creating a task graph. Optimizations can then be applied to manipulate this task graph. In our case, these task

parallel skeletons are replaced by the stream programming framework described in section 3.3, but many optimizations still hold.

The most important optimization is replacing pipelines with farms [5], shown in figure 4.5. There are two main reasons to do this optimization. First, each connection between two skeletons uses a FIFO buffer to communicate the elements, with the associated overheads. If the skeletons run on the same processor, this overhead can be eliminated. Second, there are usually more values to process than processing elements in a processor. To limit context switching, this means that in the case of a pipeline, a processor will first loop over a number of elements of the first operation before starting the next, and this limits instruction level parallelism and data locality. It is more efficient to do multiple computations on one element before moving on.

Replacing a pipeline with a composite farm is a specific case of *skeleton merging*, combining two or more skeleton calls to create a single merged skeleton, reducing overhead and improving efficiency. The two main challenges in performing this optimization are how to find the skeletons that need to be merged, and how to merge programs which, in principle, can do anything.

4.6.1 Extracting Skeleton Sequences

Merging two skeleton calls improves efficiency if the most efficient mapping maps them to the same processor. However, by combining the skeletons it is no longer possible to map them to different processors, limiting the mapping freedom. It is therefore imperative that we only merge skeletons which can be expected to run on the same processor. Additionally, skeletons can only be merged if the resulting program is functionally equivalent to the original under all circumstances. This places restrictions on the dependencies between the skeleton calls and the rest of the program.

We recognize four conditions a skeleton call sequence must satisfy before it can be merged: compatibility, colocation, control flow and data flow.

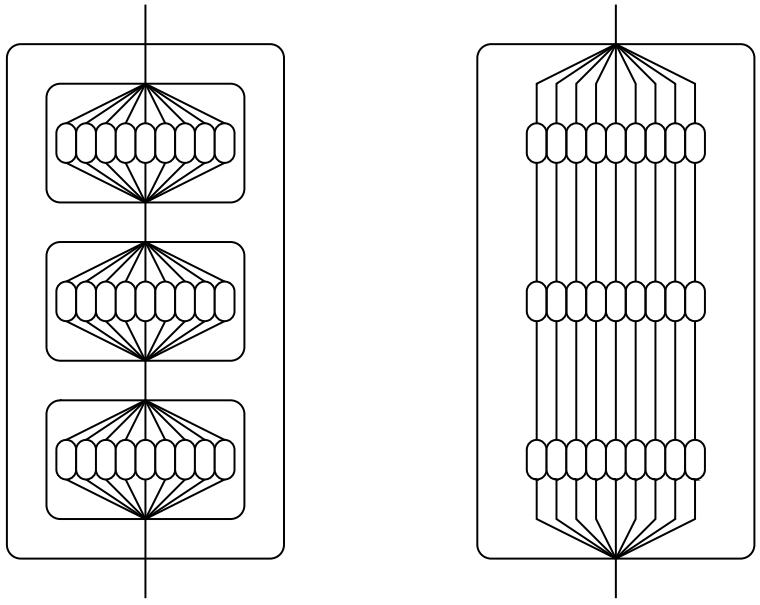
Compatibility

Since skeletons are generic programs, we can reason about their properties only in very restricted cases. However, if operations are to be efficiently merged, we need detailed information about the way in which they process their elements. This is reminiscent of the difficulties of parallelizing compilers: without limiting the properties of an operation using a skeleton, it is hard to determine the possibilities for parallelization.

We therefore restrict the generality of a skeleton by using an optional *meta-skeleton*. The meta-skeleton requires the skeleton to adhere to a certain interface which enables it to be merged. All the operations in a sequence must use the same meta-skeleton.

Colocation

In a simulation run, the operations must be mapped onto the same processor with high probability. This calculation is complicated by the fact that all operations



(a) Pipeline of data parallel skeletons.

(b) A composite farm equivalent to 4.5(a).

Figure 4.5: Replacing pipelines with a farm to improve efficiency. The buffers between tasks are removed, and each element can be processed completely without switching context before the next one is read.

will run multiple times, and we are only concerned with the location of those tasks that constitute a single sequence. As long as each separate sequence of tasks runs on the same processor, it is not prohibitive that the location of each sequence is different.

We define the *colocation probability* of an ordered pair of operations o_1 and o_2 as the average probability (measured during simulation) that a call of operation o_1 and the subsequent call of operation o_2 are mapped to the same processor. The colocation probability of a sequence of operations is then the product of all pairwise colocation probabilities between neighboring operations in the sequence.

If the colocation probability of a sequence of operations is higher than a certain threshold value, they may be merged.

Control flow

It must not be possible for the sequence to be called only partly. If the first operation is called, so must all other operations in the sequence. To ensure this, all operations must occur within the same *basic block* of the stream program; that is, there can be no control flow constructs which affect only part of the sequence.

Data flow

The data dependencies between the operations in the sequence must be static, and must be proven to be static at compile time. We perform a conservative static data flow analysis to ensure this. The connections between the operations must be streams with a constant flow of elements, and must be wholly contained within the sequence: it is not allowed for two operations to be connected via an operation outside the sequence.

Furthermore, variables produced by the sequence may not be used by statements occurring between the operations of the sequence. This is important because the merged sequence will be placed at the location of its last operation.

4.6.2 Meta-skeletons

As a skeleton provides an algorithm-specific interface for writing a certain kind of operation, limiting its expressiveness, a meta-skeleton limits the expressiveness of a skeleton. It is therefore a third-order function, receiving the code and parameters of both the skeletons and their respective kernels. In the style of section 4.5.1:

$$\begin{aligned} out &= \llbracket kernel_{\mathtt{S}} \rrbracket_{\mathtt{S}}(in) \\ &= \llbracket skeleton_{\mathtt{M}} \rrbracket_{\mathtt{M}}(kernel_{\mathtt{S}}, in) \\ &= \llbracket meta\text{–}skeleton_{\mathtt{L}} \rrbracket_{\mathtt{L}}(skeleton_{\mathtt{M}}, kernel_{\mathtt{S}}, in). \end{aligned} \tag{4.10}$$

A meta-skeleton is written in much the same way as a normal skeleton: by rewriting the skeleton and kernel bodies, and adding glue code between them. Additionally, it is the responsibility of the meta-skeleton to *statically* schedule the entire sequence, since it is now a single operation. The need for static scheduling is what requires the skeletons to be limited in their expressiveness, since data flow networks in general cannot be statically scheduled.

In the case of the `linebyline` meta-skeleton, which deals with skeletons that have a granularity of a single image line, the composing skeletons are required to consist of a `prologue` and `epilogue`, which read and allocate a statically known number of image lines, and a `repeat`ed body, processing a single line. Buffer interactions may only operate on whole lines at a time. This allows for a simple repeated schedule (all skeletons process a single line in sequence), and only requires some care during the prologue and epilogue.

We take an approach where the merged operations t communicate using light-weight line buffers. For each operation in the sequence the prologue is executed, followed by a number of normal iterations. This number of iterations is equal to the difference between the number of initial image lines needed by downstream operations (request) and the number of initial lines produced by the prologue (initial_p). In turn, the requested buffer size is equal to the number of image lines consumed by the prologue (initial_c) plus the number of iterations. A similar strategy is used for the epilogues.

$$\begin{aligned} \text{iterations}(t) &= \max_{c \in \text{consumers}(t)} \text{request}(c,t) - \text{initial}_p(t,c) \\ \text{request}(t,p) &= \text{initial}_c(t,p) + \text{iterations}(t) \end{aligned} \tag{4.11}$$

Note that because of the line granularity we do not create increased instruction-level parallelism, but only increased data locality and decreased context switching and buffer interaction overheads. Another meta-skeleton, called `pixelbypixel`, increases data locality, but it is limited to operations with pixel granularity.

4.6.3 Language features for implementing meta-skeletons

A meta-skeleton must be able to apply a skeleton to a kernel and rewrite the result. This is not possible with PEPCI as it has currently been described, because the residual code of such an application cannot be accessed by the meta-skeleton. We therefore provide a new built-in function, `reduce`, which executes a piece of code and returns the residual.

`reduce` first creates a copy of the symbol table, and executes the code using this copy. It then residualizes all variables which changed during the execution, and returns the residual code. This ensures that if the result is executed, it makes the exact same changes to the symbol table as directly executing the original code, i.e. the symbol table after evaluating the original code is that same as that after evaluating the residual:

$$\forall c \quad ST(@c) = ST(@\mathbf{reduce}(c)). \tag{4.12}$$

Another important aspect of merging skeletons is avoiding variable name clashes. PEPCI provides a library strategy to uniquely rename all local variables in a piece of code, assuring that no name clashes occur. Note that as the skeleton may introduce new variables through meta-programming, variable renaming must be applied to a reduced piece of code, and not to the original skeleton.

4.7 Results

In order to determine the validity of the skeletonization approach, we need to find out if the benefits, such as being able to use more exotic processor architectures and multiprocessor systems, outweigh the overheads. Of course, a handwritten program especially tailored to a certain application and processing architecture will always be faster than a generalized approach. However, the time and effort required to write such a program often excludes the use of certain architectures.

We will show the overhead of splitting an application into a number of kernels, and writing these kernels using skeletons. In addition, we measure the performance gain of the skeleton merging operation. All workstation experiments presented in this section were conducted using a uniprocessor backend, thus only exploiting instruction-level parallelism. Please refer to the next chapter for experiments on multiprocessor systems.

4.7.1 Skeletonization overhead

In principle it is possible to write a separate skeleton for each operation, leading to zero overhead. However, this defeats the purpose of the skeletonization approach. It indicates, though, that the skeletonization overhead is determined by how well the operation fits into the mold of the skeleton. Implementing a pixel multiplication operation using a skeleton designed for recursive neighborhood operations, while possible, is very inefficient. In general, the most restrictive skeleton that supports the operation should be chosen.

We can see in figure 4.6 that this *domain mismatch penalty* is higher for parallel architectures than for sequential ones, as is to be expected. For example, a recursive neighborhood operation on the IMAP-CE (`rn2p`) can exploit only half the parallelism of a non-recursive neighborhood operation, and induces extra overhead. In contrast, a sequential implementation suffers no extra penalty. `lutn2p`, which implements linear local neighborhood operations using a weighted kernel, suffers because of floating point conversion on the AMD Opteron. The other architectures do not support floating point arithmetic.

In many cases the XeTaL processor can not even execute an operation, because it does not have the necessary local autonomy to support those skeletons. This is another reason to use the most restrictive skeleton.

The other interesting overhead is that of splitting an application up into many separate kernels. This induces loop overhead, loss of instruction-level parallelism, and less efficient use of a processor's cache memory. Figure 4.7 shows that these effects can be quite severe, approaching a factor of two in performance.

This highlights the need to recover ILP and cache locality using skeleton merging. Our current implementation supports two meta-skeletons: `pixelbypixel`, for operations with pixel granularity (such as the pixel to pixel, anisotropic pixel and pixel lookup operations from section 2.3.2) and `linebyline` for operations with line granularity (neighborhood to pixel operations, and recursive neighborhood to pixel operations on sequential architectures). As can be seen in figure 4.7, the recovery for pixel operations is total, while for the neighborhood operations and

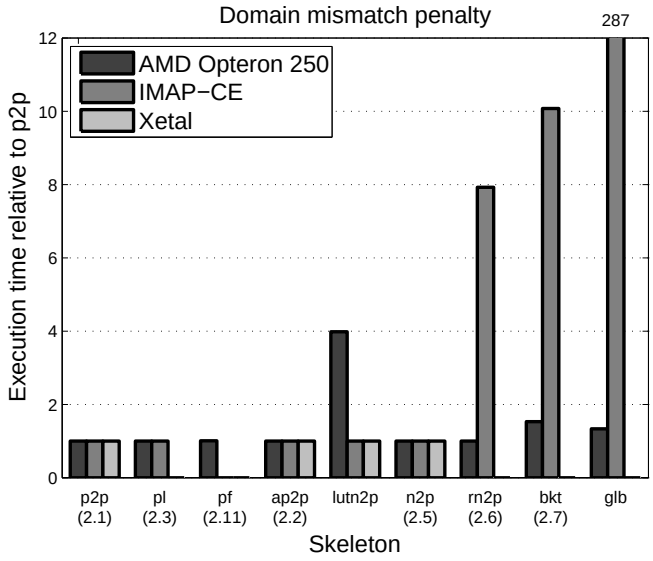


Figure 4.6: Penalty for choosing a wrong (less restrictive) skeleton for a pixel operation, relative to the execution speed of a pixel skeleton operation on the same architecture. The numbers on the X axis refer to equations in section 2.3.

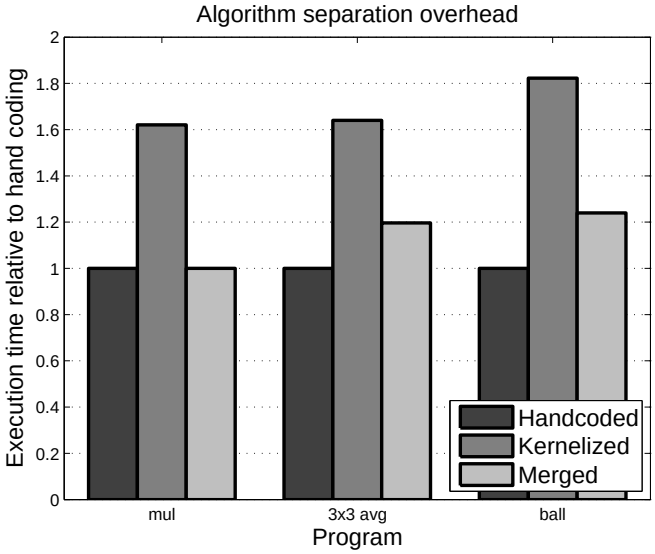


Figure 4.7: Overhead induced by separating the algorithms within an application, plotted for three programs: eight pixel multiplications, eight 3x3 neighborhood averages, and a ball following application described in [33] (a trace can be found in figure 6.3). Experiments were carried out on an AMD Opteron 250.

ball following application there remains a 20% performance loss, which we will investigate in the next section.

4.7.2 Skeleton merging performance

Figure 4.8 shows the performance benefits of skeleton merging for an image addition operation. The first two bars show that one separate image addition takes seven milliseconds for both the `PixelToPixelOp` and `NeighborhoodToPixelOp` skeletons. If we count the additional execution time taken by a second addition which is merged with the first, we see that this is almost for free with the `pixelbypixel` meta-skeleton, whereas it takes three milliseconds for the `line-byline` meta-skeleton. Merging even more operations closes the gap, but pixel merging is still more effective than line merging.

The reason for this discrepancy is twofold. First, pixel merging fuses the loops of the separate operations, reducing loop overhead, and second, it eliminates memory traffic by passing intermediate results in *registers*, whereas line merging only replaces main memory access with cache access.

The difference between merging a few operations (second group of bars) and many (third group) is the result of increased instruction-level parallelism. If a loop iteration contains more statements, more of the processor's execution units may be used; a loop iteration with only one addition is dominated by loop counter arithmetic, keeping many execution units idle. In this case, adding additional statements is free until the loop becomes dominated by the calculation instead of the looping.

We have also investigated the cache effects of skeleton merging. Context switching for operations with much internal state (such as neighborhood operations with large neighborhoods) is especially expensive due to cache trashing. While skeleton merging does avoid OS context switches, the fine-grained interleaving of operations on a line by line basis can reduce the performance gains.

Figure 4.9 shows the relative performance benefit of merging 8 neighborhood to pixel operations with increasing neighborhood sizes. The number of instructions is kept constant, but pixels are fetched from increasing distances. When accessing pixels vertically, this causes cache trashing for the larger neighborhood sizes, leading to degraded performance (as the image is stored in row-major order).

The control case, using horizontal access, also shows a decrease in performance, although of a different order than that of the vertical access. This is due to border handling and other inefficiencies due to merging.

4.8 Discussion

We have introduced a language and tool, called PEPCI, for implementing skeletons based on pseudo-dynamic meta-programming and embedded term rewriting. The pseudo-dynamic approach allows the mind set of writing a program to directly process some input while actually *generating* such a program, and is achieved using partial evaluation. This makes it easy to implement the structure of the skeleton. Embedded term rewriting, using the Stratego term rewriting language,

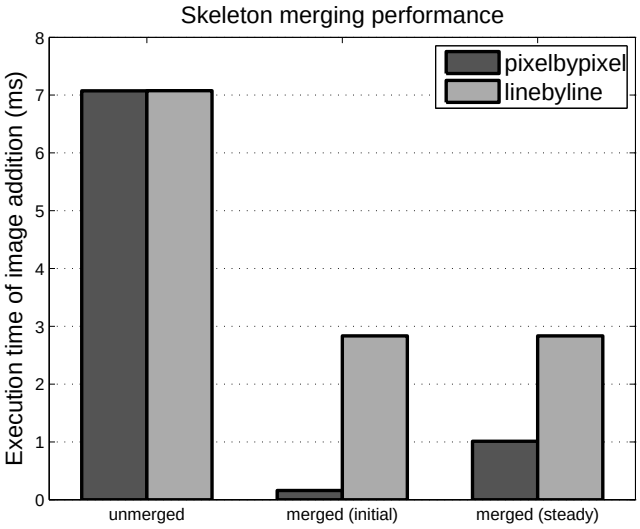


Figure 4.8: Performance of pixel and line merging meta-skeletons for image addition. *initial* means the added cost of merging a second addition, and *steady* is the steady state cost of merging reached after a number of additions.

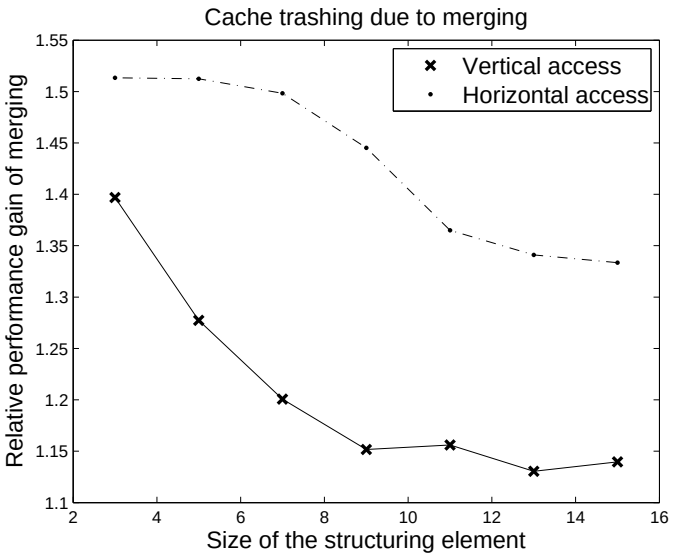


Figure 4.9: `linebyline` skeleton merging performance as a function of the neighborhood size. The number of instructions for each measurement is constant, but the distance from which pixels are accessed is different.

allows both simple pattern substitution (through library calls) and more involved program analyses and transformations (by writing inline Stratego programs).

The entire PEPCI language is based around the concept of layered complexity. PEPCI programs may range from simple ANSI C (if no meta-programming is required, and just the speedup provided by partial evaluation is desired) to elaborate Stratego programs directly modifying abstract syntax trees; it is not necessary to understand a language concept if it is not needed in the program.

We introduced the concept of meta-skeletons, which limit the expressivity of skeletons in order to allow us to reason about their execution, especially in order to do merging and static scheduling to reduce overhead. Using skeleton merging, the single-processor performance of a properly skeletonized application is only around 20% slower than a dedicated handcrafted version.

Traditional languages using algorithmic skeletons [116, 46, 97] treat them as black-box components, unable to be modified by the user. This restricts their flexibility. On the other hand, the skeletons are generally *compositional* (able to be nested in arbitrary ways), and these compositions may be optimized by the compiler. We do not support skeleton nesting, instead relying on the extensibility of our skeleton library: if the provided patterns do not suffice, a (specialized) programmer can add his own.

Chapter 5

Implementing stream programming using RPC

Stream programming is a way of writing architecture-independent parallel image processing applications. For our application domain, we have determined that we wish to use imperatively connected finite, multidimensional streams. Furthermore, the stream kernels should be mapped to a heterogeneous system and executed as much in task parallel as possible. This task parallelism should continue across dynamic stream reconfigurations.

In chapter 4 we presented our method of exploiting data parallelism using algorithm-specific languages as skeletons. Data parallelism is the main source of parallelism in image processing applications. The subject of this chapter is how the instantiated operations are interconnected to exploit task parallelism across a multiprocessing system. Our approach to this problem is based on the *Remote Procedure Call* (RPC) mechanism known from distributed programming.

The basic method, with several extensions needed to improve parallelism, is described in section 5.1. Section 5.2 describes the run-time environment that permits stream programming using RPC, while sections 5.3-5.4 present the various models needed by the run-time environment to find an appropriate mapping of kernels to processors. The management of the stream buffers is presented in section 5.5, and finally section 5.6 shows results verifying the amount of overhead and achieved parallelism.

5.1 Remote procedure call

Remote procedure call (RPC) [138, 15] was introduced as a way of standardizing command/response protocols in distributed systems. It extends the idea of function calls to remote functions, preserving the well-understood mechanics behind local function calls. An RPC run-time environment (RTE) consists of a *client* program, containing the RPCs, and a *server* program, running on a remote system. The RPC functions provided to the client are *stubs* that perform the following acts:

- Copy the arguments of the stub function to the server.
- Cause the server program to jump to a specific procedure, depending on the *RPC identifier* of the stub.
- Wait until the remote procedure finishes execution.
- Read the result of the remote procedure, and return it as the result of the stub.

Thus, the flow of control passes from the client to the server, just as it normally passes from caller to callee in a single computer. Note that the remote function lives inside another address space than the client, and therefore cannot access global variables. All necessary information has to be passed using arguments.

Program 5.1 shows an example C++ RPC client program. We have bindings for both C and C++ as the client language, as they are widely used in the embedded systems world, and therefore provide for an easy migration path. All arguments are passed by reference. A data flow graph of the interactions between the functions is depicted in figure 5.1.

Program 5.1 Example RPC client program

```
//                    In  In  Out
localfunctionS      (          &s );
remoteFunctionA(&s,            &a );
remoteFunctionB(&a,            &b );
remoteFunctionC(&b,            &c );
localFunctionD      (&a,       &d );
remoteFunctionE(&b, &d,        &e );
remoteFunctionF(&c, &e,        &f );
localFunctionQ      (&f           );
```

The Gantt chart [57] in figure 5.2(a) indicates how the flow of control passes between client and server. In this basic, synchronous, implementation of RPC, the client is waiting for the server to finish processing before proceeding. An *asynchronous* implementation hands control back to the client immediately, and returns a token that can be used to poll whether the computation has finished. This allows the client to continue executing until the result is needed, see figure 5.2(b).

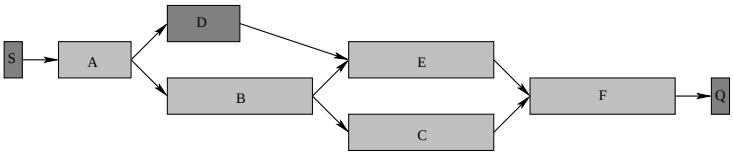
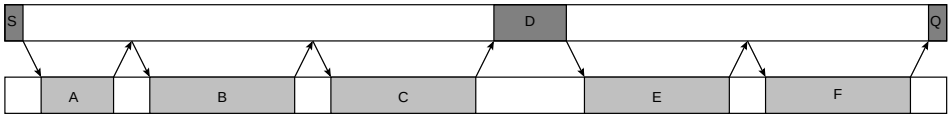
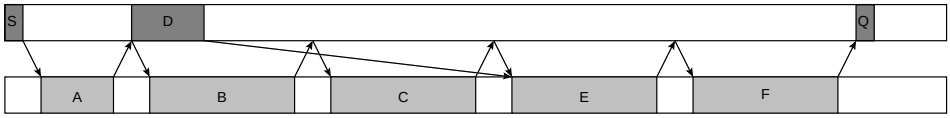


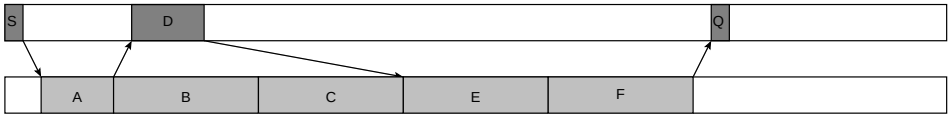
Figure 5.1: Task graph belonging to program 5.1. Darkly colored boxes are local functions.



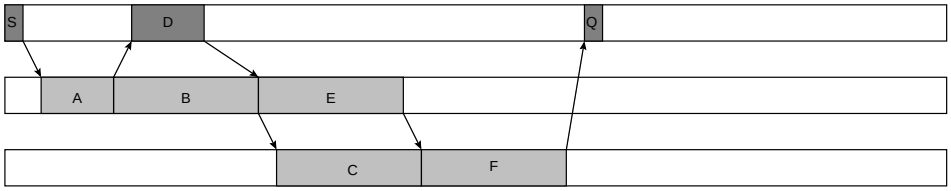
(a) Synchronous RPC. The client program waits for the completion of each remote function.



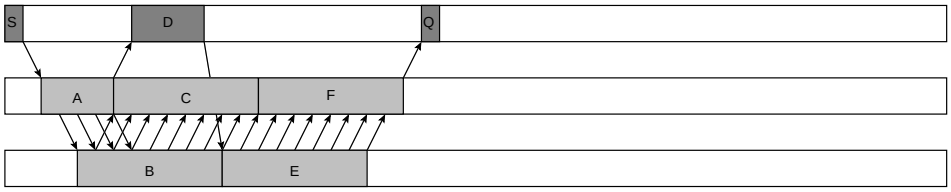
(b) Asynchronous RPC. Note how the client can continue with D while the server is still processing B.



(c) Asynchronous RPC using batched futures. Message latencies are avoided by sending batches of calls to the server, and allowing them to reference each other's results.



(d) Using futures across multiple servers. The results of a remote function are directly copied to the servers that need them.



(e) Using series of partial results allows us to pipeline computations. `B` can start after receiving the first partial result of `A`. `C` can start after receiving the first partial result of `B`, etc. The local functions are assumed to be non-pipelined.

Figure 5.2: Gantt charts for the execution of program 5.1 using increasingly efficient implementations of remote procedure call (RPC).

5.1.1 Future-based RPC

If the token returned by an RPC call is an object that can be used to access the result of the call (as opposed to simply relaying that some result has been written to a buffer), it is also called a *future* [12] or *promise* [88]. Futures were introduced, not within the context of RPC, but as a parallel, eager evaluation method for functional programming languages in a shared-memory environment. For example, in MultiLisp [66], futures could be used as normal variables, automatically blocking the program if they were needed for a computation.

A first application of futures in a distributed environment can be found in [88]. However, unlike their shared-memory counterparts, they could not be passed to other RPC functions without first being *claimed* by the client program. As a result, the client had to wait for an earlier RPC call to finish before passing the result to a next call. In order to achieve parallelism, explicit threading had to be used.

An extension, called *batched futures* [103], was designed to allow RPC calls to reference each other's results. The main reason for this extension was not to exploit parallelism, but to avoid the latencies involved in bouncing messages back and forth between client and server by sending them in *batches*. This is illustrated in figure 5.2(c).

We can extend the concept even further by allowing futures to be used across servers [7]. This introduces task parallelism between multiple servers, even if some computations are dependent (see figure 5.2(d)). An even larger degree of parallelism can be achieved if the remote functions generate a series of partial results, allowing the use of *pipelining* (figure 5.2(e)).

With the addition of pipelining, we have now gained an RPC implementation that satisfies all the requirements for a highly task parallel, imperatively connected stream programming language. We can dynamically construct arbitrary acyclic graphs of RPC calls and execute them on a heterogeneous multiprocessor system, while retaining the semantics of sequential function calls.

5.1.2 Encapsulating local functions

A problem remains when the shape of the task graph itself depends on the result of a remote function. In this case, further construction of the task graph is halted until the offending future has been resolved, even those calls which do *not* depend on that future. To solve this, we allow local functions to be encapsulated and called as remote functions. This causes them to run asynchronously, so that further task graph construction may continue. However, to ensure determinism it is required that such asynchronous local functions – like the stream kernels themselves – have no interacting side effects.

In program 5.2, `localDecisionFunction` decides whether to call `remoteFunctionC` or `remoteFunctionD`, based on some local computation `localFunctionB`. It is encapsulated using `futurecall` [109] so that the `while` loop can continue iterating, allowing more pipelining. This results in an incomplete task graph, since the part between `a` and `cd` is missing. Once the local computation finishes and the task graph is completed, execution of it can continue.

Program 5.2 Encapsulating dynamic task graph construction.

```
void localDecisionFunction(*a, *cd)
{
  if (localFunctionB(a))
    remoteFunctionC(a, cd)
  else
    remoteFunctionD(a, cd)
}

while (1)
{
  remoteFunctionA(&a);
  futurecall(localDecisionFunction, &a, &cd);
  remoteFunctionE(&cd);
}
```

5.2 Run-time environment

A run-time environment's tasks are: keeping track of the futures, mapping remote function calls to servers, and managing stream transportation. It consists of the following parts, which are schematically shown in figure 5.3:

- A *frontend*, which records all remote function calls as well as encapsulated local function calls. It tracks futures to create the application's task graph, and blocks on unresolved futures. The frontend presents the run-time environment to the user.
- A *mapper* that selects among the set of possible mappings of the task graph to the architecture, with the help of
- an *evaluator* which predicts the performance of a particular mapping.
- A *dispatcher* which determines the needed stream buffer sizes and allocates the necessary buffers and transports before dispatching the functions.
- A *gatherer*, collecting information about function completion, and signalling future resolution to the frontend.

During kernel extraction and skeleton instantiation (see chapter 4), stubs are generated for each kernel call, with corresponding server functions on all processors which support the operation. These are linked by an *RPC ID*. The run-time environment itself is a standard C-library, with a standard C and C++ API.

5.2.1 API

Our C++ API introduces a Future template class which implements the future semantics for any data type. These scalar futures may be assigned to and used in any expressions which require the original data type. The only difference is

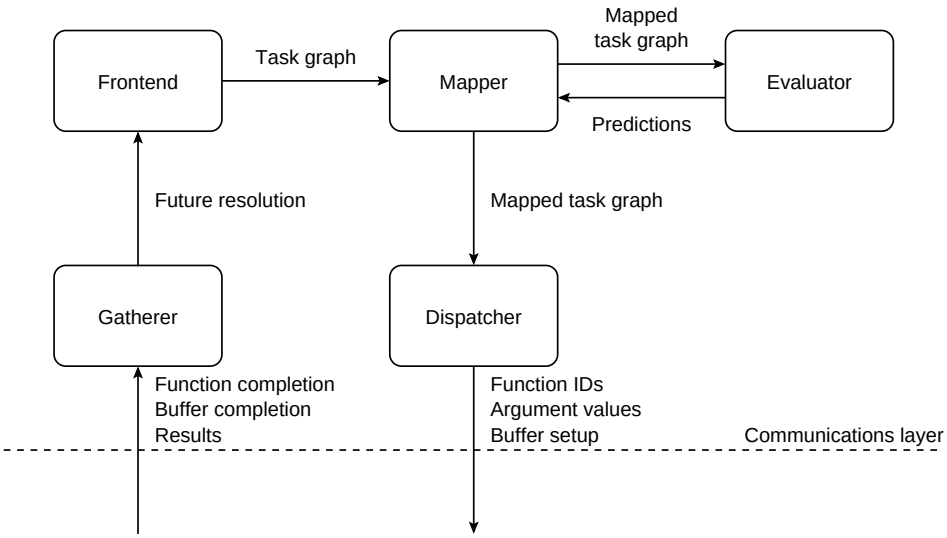


Figure 5.3: Components and data flow within the RPC run-time environment.

that a check is made to ensure that the future is resolved before the expression is computed.

Streams must first be loaded into a standard C-array before their contents may be accessed. They are dynamically typed, and instantiate the Future class using the STREAM data type.

Program 5.3 Iterative diffusion until idempotence using future-based RPC with a C++ API.

```
1   Future<STREAM> im, diff, absd;
2   Future<int> s;
3
4   capture(&im);
5   do
6   {
7     diffuse(&im, &diff);
8     absdiff(&im, &diff, &absd);
9     sum(&absd, &s);
10    im = diff;
11  } while (s > THRESHOLD);
```

Program 5.3 implements an iterative diffusion algorithm. A captured image is subjected to diffusion until the difference between two successive steps is smaller than a certain threshold. Note that assigning a future to another future (such as in line 10) does not block the program, because the actual value is not used. When it *is* used, like in line 11, the program blocks until the future is resolved.

5.2.2 Tracking futures

Futures are tracked by their interactions with RPC stubs or local functions called using `futurecall`. They are used to set up a task graph such as in figure 5.1: each RPC stub creates a task vertex, and each use of a future creates an edge. Such task graphs are always acyclic, as it is impossible for an operation to reference an input future which has not been produced yet.

It is important to note that futures allow us to provide sequential call-by-value semantics in an asynchronous environment. When a future is overwritten, it does not change the task graph edges that have already been created. Instead, any further references will simply use a new task graph vertex as the origin. As a previously created part of the task graph is not changed by later calls, the result is the same as if every operation finished execution before returning.

When a future is copied, such as when it is assigned to another future or when it is passed as an argument of an encapsulated local function, a new *reference* is created. These references may be used and overwritten independently, so that they may eventually contain different values or point to different origins.

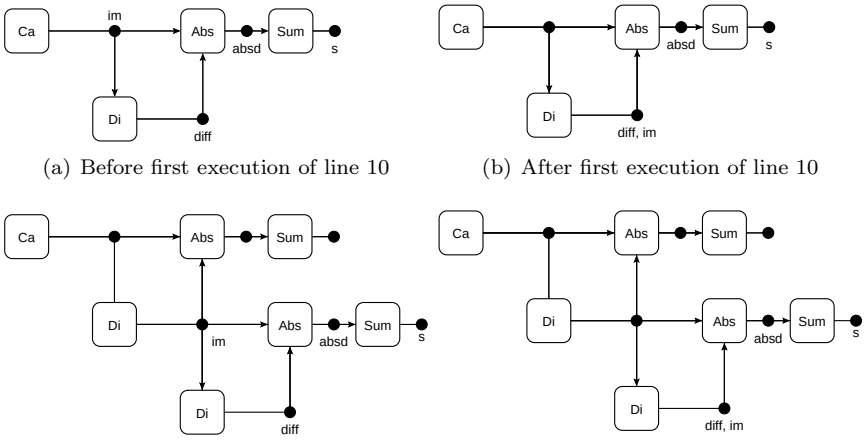
Figure 5.4 shows four stages in the execution of the iterative diffusion program 5.3. Between 5.4(a) and (b), `im` is set to `diff` (line 10), thereby pointing to a different origin but leaving the original task graph intact. During a later reassignment of `diff` (in the second execution of line 7), they again point to different origins (5.4(c)).

The *gatherer* is responsible for writing the correct values into resolved scalar futures. A value is only written into futures which still reference the produced value. If no futures reference the result anymore, it is disposed. Resolved streams are not automatically written to a future, because transporting all stream data to the control processor would create a communications bottleneck. Instead, the contents of a stream must be explicitly loaded into an array if they are needed in the client program.

5.3 Mapping

The mapper has to find an efficient mapping of remote functions to servers. It does this by constructing different mappings of the task graph to a model of the architecture, and passing these to the evaluator. We will first describe the architecture model and related information. Then, we will more formally describe the task graph that is created by the frontend, called the *stream task graph*. Finally, we will describe how the stream task graph is transformed by mapping it to the architecture model, creating the *Dependent Task Interaction Graph*.

The mapping itself is based on a simple list scheduling technique with topological ordering. As such, in algorithm 5.1, **priority** is simply the order in which tasks arrive at the mapper, while **objective** is the predicted execution time. If time permits, the scheduling process can be iterated for a more accurate prediction (because **objective** may then include tasks that were scheduled in the previous iteration).



(a) Before first execution of line 10

(b) After first execution of line 10

(c) Before second execution of line 10

(d) After second execution of line 10

Figure 5.4: Different stages in task graph construction for the iterative diffusion in program 5.3

.

Algorithm 5.1 List scheduling. **T** is a set of tasks, while **A** is a set of targets.

for $i \leftarrow 1 \ldots \|\mathbf{T}\|$
  choose $t \in \mathbf{T}$ which maximizes **priority**
  schedule t on $a \in \mathbf{A}$ which minimizes **objective**
  remove t from **T**

5.3.1 Architecture model

The architecture model specifies which servers (*targets*) are available, and how they are connected. Communication channels are modeled as servers as well, but they can only run *transport* operations.

Figure 5.5 shows a model of the Inca+ architecture [80], consisting of a sensor, XeTaL SIMD and TriMedia VLIW processor. The square boxes are communication channels, while the rounded boxes are targets that can perform an operation other than transferring information. The arrows are the links connecting the various targets.

There are three channels going from the XeTaL to the TriMedia: one for each of red, green and blue. These channels can only transport one stream at a time. This is modeled by assigning each channel a *stream* resource, and having each *transport* operation require such a resource. Mappings which violate the resource constraints are automatically discarded.

Resources can also be used to model other constraints, such as a limited amount of memory or the fact that a camera can only take one picture at a time. An architecture model is therefore a directed graph, labeled with the presence of resources.

Definition 5.1 (Architecture model) *An architecture model AM is a tuple* $(\mathbf{A}, \mathbf{L}, \mathbf{R}, pres)$*, where:*

- $\mathbf{A}$ *is a set of target vertices*
- $\mathbf{L} \subseteq \mathbf{A} \times \mathbf{A}$ *is a set of link edges.* $(\mathbf{A}, \mathbf{L})$ *form a directed graph.*
- $\mathbf{R}$ *is a set of resources.*
- $pres : \mathbf{A} \times \mathbf{R} \to \mathbb{N}$ *defines the number of resources* $r \in \mathbf{R}$ *a target* $a \in \mathbf{A}$ *provides.*

Note that an RPC system only exploits task parallelism. Data parallelism is exploited by executing the operations on data parallel targets; such targets (which could be static aggregations of sequential processors) are single vertices in the architecture model.

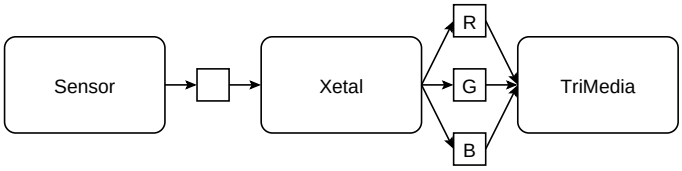


Figure 5.5: Inca+ architecture model. Note that data can flow in only one direction: from sensor to XeTaLto TriMedia.

5.3.2 Application specification

During compilation, information is extracted about the application, and entered into an *application specification*. This data is gathered in four stages of the compilation process.

Kernel extraction

The kernel extraction provides a set of available remote function *operation types* (associated with RPC IDs), as well as the intent of an operation's argument: whether they will be used as input or output. This is used during future tracking.

Skeleton instantiation

As described in the previous chapters, skeletons are implemented for different processor architectures. A skeleton compiling a kernel can therefore provide architecture-specific information about the resulting operation. This information consists of the number of resources that an operation requires of a certain target, some information about argument buffer sizes (such as was used in section 4.6), and the distribution of an argument's stream elements over time.

The stream element distribution can be either *regular* or *bulk*. An operation argument is called regular if the argument's stream elements produced or consumed by the operation are spaced regularly with respect to the operation's use of CPU time. It is called bulk if all the stream's elements are consumed before the operation starts, or of all the stream's elements are produced only after the operation finishes. We treat scalar arguments, which have only one element, as bulk.

Trace generation

A simulation of the entire application generates a *trace* of the data flow, providing the maximum stream length an operation will produce, which is important for determining buffer sizes. We will assume that the simulation runs sufficiently long, so that all dynamic paths through the program are taken, and the stream lengths are real maxima. The trace generation is described in more detail in section 6.2.1

Benchmarking

After the trace generation, each operation is benchmarked individually on each architecture it can run on, giving the CPU time it needs to complete its task. This will be described in section 6.2.2.

Definition 5.2 (Application Specification) *An application specification AS is a tuple* $(\mathbf{O}, \mathbf{OA}, intent, rres, dist, len, time)$*, where:*

- $\mathbf{O}$ *is the set of available operation types.*
- $\mathbf{OA} \subseteq \mathbf{O} \times \mathbb{N}$ *is the set of operation arguments.*
- $intent : \mathbf{OA} \to \{in, out\}$ *is the intent of each operation argument.*

- $rres : \mathbf{O} \times \mathbf{A} \times \mathbf{R} \to \mathbb{R}$ *is the amount of required resources for an operation on a specific target.* $\mathbf{A}$ *and* $\mathbf{R}$ *are taken from the architecture model.*
- $dist : \mathbf{OA} \times \mathbf{A} \to \{regular, bulk\}$ *is the distribution of stream elements for an operation argument on a target.*
- $len : \mathbf{OA} \to \mathbb{N}$ *is the maximum length of a stream produced by an operation argument.*
- $time : \mathbf{O} \times \mathbf{A} \to \mathbb{R}$ *is the execution time of an operation on a target. For transport operations, it specifies the time it takes to transfer a single byte.*

The architecture model and application specification are passed to the run-time system using a set of XML files, the syntax of which is detailed in appendix A.

5.3.3 Stream task graph

The stream task graph (STG) is formed during program execution by future tracking. It is a directed acyclic graph interleaving tasks and streams. Enqueueing a remote function generates a task vertex, creating a new stream vertex and producer edge for each output. For all futures the function references as inputs, consumer edges are created from the appropriate stream to the task.

Figure 5.6 is an example stream task graph for an edge detection program. The graph is annotated using the information gathered during compilation: tasks are assigned the appropriate operation type, producer and consumer edges are annotated with their stream distribution, and streams have a length based on the maximum length of their producers.

Definition 5.3 (Stream task graph) *A Stream Task Graph STG is a tuple* $(\mathbf{T}, \mathbf{S}, \mathbf{P}, \mathbf{C}, optype, dist, len)$*, where:*

- $\mathbf{T}$ *is the set of task vertices.*

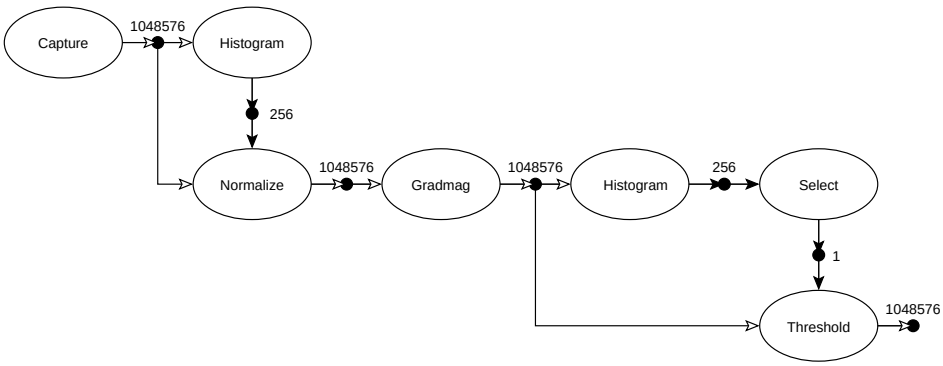


Figure 5.6: Edge detection stream task graph. Regular edges have white arrows, while bulk edges are black. The streams are annotated with their lengths.

- $\mathbf{S}$ *is the set of stream vertices.*
- $\mathbf{P} \subseteq \mathbf{T} \times \mathbf{S}$ *is the set of producer edges. A stream cannot have more than one producer:*
$$(t, s) \in \mathbf{P} \wedge (u, s) \in \mathbf{P} \implies t = u \tag{5.1}$$
- $\mathbf{C} \subseteq \mathbf{S} \times \mathbf{T}$ *is the set of consumer edges.*
- $optype : \mathbf{T} \to \mathbf{O}$ *is the operation type function.* $\mathbf{O}$ *refers to the application specification.*
- $dist : \mathbf{P} \cup \mathbf{C} \to \{regular, bulk\}$ *is the stream element distribution function, equivalent to the dist of the corresponding operation argument.*
- $len : \mathbf{S} \to \mathbb{R}$ *is the stream length function, equivalent to the len of the operation argument producing the stream.*

5.3.4 Task and stream mapping

A mapping is now an assignment of the tasks in the stream task graph to the targets in the architecture model. If the targets are not directly connected, a shortest path is found through the intermediates, implying transport operations of the stream's length. To spare computation time, this path is static except where it would cause resource over-commitment.

Definition 5.4 (Mapping) *A task mapping* $m : \mathbf{T} \to \mathbf{A}$ *is a function assigning a target* $a \in \mathbf{A}$ *to each task* $t \in \mathbf{T}$*. A task mapping implies a stream mapping* $m_s : \mathbf{S} \to 2^{\mathbf{A}}$*, of streams to the targets they need to hit. The stream mapping is associated with a path function* $m_p : \mathbf{S} \to 2^{\mathbf{A} \times \mathbf{A}}$ *detailing the spanning tree needed to communicate the stream across all targets in* $m_s(s)$*. Here,* $2^{\mathbf{A}} = \{\mathbf{A}' | \mathbf{A}' \subseteq \mathbf{A}\}$ *is the* powerset *of* $\mathbf{A}$*.*

The mapping transforms the stream task graph into a *dependent task interaction graph (DTIG)*, which is used by the evaluator.

Definition 5.5 (Dependent Task Interaction Graph) *A DTIG is a directed acyclic graph representing a mapped stream task graph. It is a tuple* $(\mathbf{V}, \mathbf{E}, \mathbf{A}, m, w, d)$*, where:*

- $\mathbf{V}$ *is a set of task vertices, which include stream transport operations.*
- $\mathbf{E} \subseteq \mathbf{V} \times \mathbf{V}$ *is a set of communication edges.* $(\mathbf{V}, \mathbf{E})$ *form a directed acyclic graph.*
- $\mathbf{A}$ *is a set of targets, the same as in the architecture model.*
- $m : \mathbf{V} \to \mathbf{A}$ *is the task mapping function.*
- $w : \mathbf{V} \to \mathbb{R}$ *is the task cost function, specifying the required computation time of task* v *on target* $m(t)$*.*

- $d : \mathbf{E} \to \{interacting, dependent\}$ *is the stream element distribution function, specifying whether an edge is* interacting *(connecting operation arguments which are both* regular*) or* dependent *(if either the source or destination operation argument is* bulk*).*

The DTIG combines a task dependency graph (TDG) and a task interaction graph (TIG) by having the d function specify which edges are dependency edges and which are interaction edges. Dependencies arise when an operation needs to finish before it produces its data, as opposed to the continuous generation of stream elements in a regular stream connection.

The DTIG is constructed as in algorithm 5.2. It starts by creating a DTIG task vertex for every STG vertex, with cost as in the relevant operation type in the application specification. Next, for each stream, it creates a DTIG task vertex for every processor it passes through. These are transport operations, and are assigned a computation time depending on the stream length. It then creates the edges from the producing tasks to the transport operations and from the transport operations to the consuming tasks. Finally, it interconnects the transport operations based on the stream's spanning tree. If a stream passes through only one target, no transport operation is created. Local communication is therefore modeled as being infinitely fast.

Figure 5.7 shows how the stream task graph of figure 5.6 can be mapped onto an architecture to create a DTIG.

5.4 Performance prediction

A cost model is a set of assumptions about how a DTIG executes. Based on these assumptions, the evaluator predicts the expected completion time, or *makespan.*

Definition 5.6 (Makespan) *The* makespan $ms : \mathbf{DTIG} \to \mathbb{R}$ *of a Dependent Task Interaction Graph DTIG is the time it takes for all tasks* $v \in \mathbf{V}$ *to finish executing.*

Because the DTIG is constructed as a mapping of a stream task graph onto an architecture model under a certain application specification, its makespan is the expected completion time of the application itself.

We will describe two cost models: independent and dependent. Both have the following common assumptions:

Assumption 5.7 (Overhead-free task switching) *Executing multiple tasks concurrently on a target a does not incur any overhead. That is, the makespan of two independent tasks* v *and* v' *on* a *is equal to the sum of the execution times of* v *and* v'*, regardless of their interleaving.*

Corollary 5.8 (Continuity of concurrent processes) *As tasks can be switched arbitrarily fast without overhead, we can model a task* $v \in \mathbf{V}$ *allocated a fraction* $f_{m(v)}(v) = \epsilon$ *of CPU time on* $m(v)$ *as a continuously running process taking* $w(v)/\epsilon$ *seconds to complete.*

These assumptions make it possible to reason about task execution times in a continuous way, without requiring us to predict the exact interleaving of tasks.

Algorithm 5.2 Algorithm to generate the DTIG from a stream task graph

```
Create a vertex for each task
V ← ∅
∀t ∈ T
  v ← new task
  V ← V ∪ v
  m(v) ≡ m(t)
  w(v) ≡ time(optype(t), m(t))
  taskmap(t) ≡ v

Create edges
∀t ∈ T
  Loop over streams produced by this task
  ∀p = (t, s) ∈ P
    Check if we need to set up remote communication
    if ‖m_s(s)‖ > 1
      Create a vertex for each target the stream passes through
      ∀a ∈ m_s(s)
        v ← new task
        V ← V ∪ v
        m(v) ≡ a
        w(v) ≡ len(s) · time(transport, a)
        targetmap(a) ≡ v

        Create edge from producer task to producer target
        if m(t) = a
          e ← (taskmap(t), v)
          E ← E ∪ e
          d(e) ≡ dist(p)
        Create edges from consumer target to consumer tasks
        ∀c = (s, t′) ∈ C
          if m(t′) = a
            e ← (v, taskmap(t′))
            E ← E ∪ e
            d(e) ≡ dist(c)
        Create transport edges
        ∀(a, a′) ∈ m_p(s)
          e ← (targetmap(a), targetmap(a′))
          E ← E ∪ e
          d(e) ≡ interacting
    else
      Create edges from producer task to consumer tasks
      ∀c = (s, t′) ∈ C
        e ← (taskmap(t), taskmap(t′))
        E ← E ∪ e
        d(e) ≡ { interacting   if dist(p) = regular ∧ dist(c) = regular
               { dependent    otherwise
```

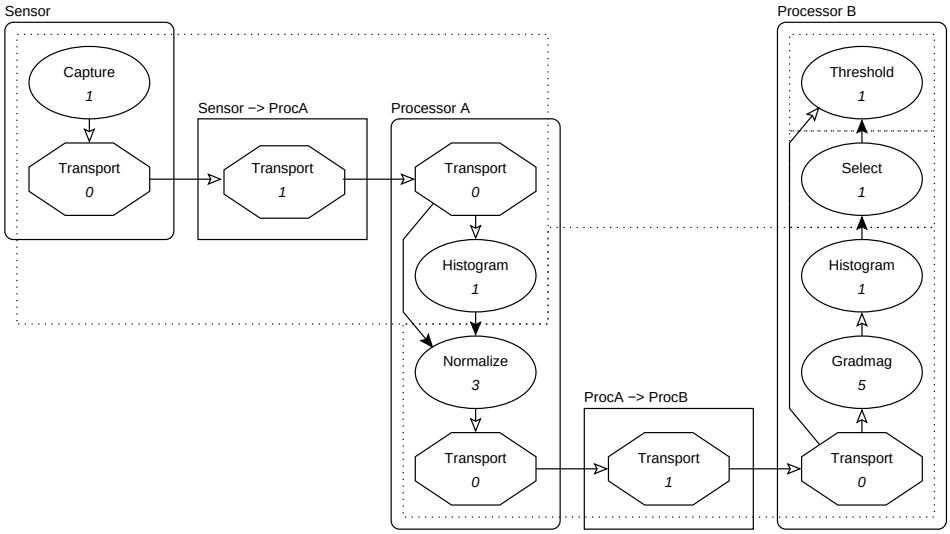


Figure 5.7: DTIG for the mapped edge detection stream task graph. The architecture consists of a sensor and two processors, in a pipeline. Communication is modeled to be free on the processors ($w(v) = 0$ for transport operations on "real" processors). Open arrows are interacting, while closed arrows are bulk edges.

5.4.1 Performance model for interacting tasks

Our least expensive performance model assumes there are no dependencies; $\forall e \in \mathbf{E} : d =$ interacting; the DTIG thus collapses into a standard TIG. First, consider a *dtig* without any edges at all:

Theorem 5.9 (Makespan of independent tasks) *Assume that* $\mathbf{E} = \varnothing$*. Then,*

$$ms(DTIG) = \max_{a \in \mathbf{A}} \left(\sum_{\{v \in \mathbf{V} | m(v) = a\}} w(t) \right). \tag{5.2}$$

Proof There are no task interactions. Therefore, the processors do not have to wait on each other. The total completion time is then the completion time of the processor with the largest amount of work assigned to it.

Note that even if we have not modeled task interactions, we *have* modeled the communication explicitly as tasks, by virtue of the construction of the DTIG. In this model, the makespan of the DTIG in figure 5.7 is 8 (from processor B).

In order to model the execution time in light of task interactions, we need to model the task interactions themselves. We assume interacting tasks to run in *lock-step*.

Assumption 5.10 (Lock-step execution) *Interacting tasks run in lock-step. That is, for two interacting tasks* $v, v' \in \mathbf{V}$ *running on target* $a \in \mathbf{A}$*,* v'*'s CPU*

time fraction is

$$f_a(v') = \frac{w(v')}{w(v)} f_a(v). \tag{5.3}$$

In a streaming environment, with different tasks operating on different parts of a long stream, this is a reasonable assumption because the communication latency is amortized over the entire stream.

Theorem 5.11 (Makespan of interacting tasks) *The makespan of a set of interacting tasks is the same as the makespan of a set of independent tasks.*

Proof There will be one processor with the highest total load:

$$l_{max} = \max_{a \in \mathbf{A}} \left(\sum_{\{v \in \mathbf{V} | m(v) = a\}} w(t) \right). \tag{5.4}$$

By assumption 5.10, $\forall v \in \mathbf{V} : f_{m(v)}(v) = s_v \cdot w(v)$ for some constant s_v. As the maximum load of a processor is one, and there are no other restrictions on the execution speed, s_v must be $1/l_{max}$: $f_{m(v)}(v) = w(v)/l_{max}$. By corollary 5.8, we can model these processes as running *at the same time*. Therefore, all tasks will finish after l_{max} seconds.

Under an optimal schedule, any TIG has the same makespan as a connected TIG. Even though assumption 5.10 does not apply for tasks that do not interact, we can still assume a schedule which satisfies it.

5.4.2 Performance model for dependent tasks

To model dependencies, we use assumption 5.10 to generate a hypertask dependency graph (HDG), with vertices (hypertasks) that require work on *multiple* targets. It is created from the DTIG by grouping sets of interacting tasks into hypertasks, eliminating the interaction edges. As such, only dependency edges remain between the hypertasks.

Definition 5.12 (Hypertask Dependency Graph) *An HDG is represented as a tuple* $(\mathbf{H}, \mathbf{A}, WORK, DEP)$*, where*

- $\mathbf{H}$ *a set of hypertasks.*
- $\mathbf{A}$ *a set of targets.*
- $WORK$ *is a* $\|\mathbf{H}\| \times \|\mathbf{A}\|$ *matrix.* $work_{h,a}$ *is the amount of work required by hypertask* h *on target* a*.*
- DEP *is a* $\|\mathbf{H}\| \times \|\mathbf{H}\|$ *matrix.* $dep_{h,h'}$ *is 1 iff* h *is dependent on* h'*. DEP encodes a directed acyclic graph.*

The hypertasks are generated from the DTIG by finding groups of tasks that are connected only by interaction edges. We first annotate the DTIG vertices with the maximum distance (in dependencies) from an upstream node; if two

tasks then have the same distance, we can be sure that there is no dependency edge in any path between them. Next, we label each connected set of tasks that have the same distance with the same label, thereby finding maximal interacting subsets. All the work of the tasks in each subset is then added to get the work for the hypertasks; as the tasks are not necessarily mapped to the same processors, the hypertasks may therefore require work on more than one processor. Finally, any dependencies between tasks in different subsets are promoted to dependencies between the hypertasks themselves. See algorithm 5.3.

Algorithm 5.3 Algorithm to generate a HDG from a DTIG. $S^{\leftrightarrow}$ means the commutative closure of $\mathbf{S}$: $(a, b) \in \mathbf{S} \implies (b, a) \in \mathbf{S}$

```
Find distances
for i ← 1 ... ||V||
  v ← topologicalorder(V, i)
  distance(v) ← max_{(v',v)∈E} distance(v') + ld
    where ld = { 1  if d(e) = dependent
               { 0  otherwise

Assign labels
for i ← 1 ... ||V||
  v ← topologicalorder(V, i)

  if ¬label(v)
    l ← new label
    label(v) ≡ l

    Recursively propagate label over unlabeled vertices with
    the same distance
    recurse(x; ∀(v, v') ∈ E^↔
      if ¬label(v') ∧ distance(v) = distance(v')
        label(v) ≡ l
        x(v'))

WORK = 0
DEP = 0

Add work of all tasks with the same label
∀v ∈ V
  work_{label(v),m(v)} ← work_{label(v),m(v)} + w(v)

Promote dependencies between tasks to dependencies between
hypertasks
∀(v, v') ∈ E
  if d(e) = dependent
    dep_{label(v'),label(v)} ← 1
```

Figure 5.8 shows the resulting *WORK* and *DEP* matrices for the edge detection DTIG of figure 5.7. All tasks within each dotted region of that figure are grouped together into a hypertask. To find the makespan of an HDG, we do not assume an optimal schedule, instead modeling a local preemptive scheduler. We can model the local scheduler for independent DTIG tasks by requiring all tasks on a target to have the same CPU time fraction:

$$\forall a \in \mathbf{A}, v \in \mathbf{V} : f_a(v) = \frac{1}{\|\{v' \in \mathbf{V} | m(v') = a\}\|}. \tag{5.5}$$

However, HDG hypertasks interact, and may be limited because of the execution time allocated to them on another target (since they must run in lock-step). Define the *speed* s_h of a hypertask h such that

$$\forall a \in \mathbf{A} : f_a(h) = s_h \cdot work_{h,a}. \tag{5.6}$$

One way to satisfy equation 5.5 is by setting s_h to the *minimum* of all restrictions:

$$s_h = \min_{\{a \in \mathbf{A} | work_{h,a} > 0\}} \frac{1}{\|\{h' \in \mathbf{H} | work_{h',a} > 0\}\| \cdot work_{h,a}}. \tag{5.7}$$

However, this is a very pessimistic assumption, as a local scheduler will not reserve a specific time slice for each process, but rather allow another process to continue if one process does not use its entire time slice. Suppose $\mathbf{s}$ is an initial speed vector for the hypertasks. Algorithm 5.4 will restrict $\mathbf{s}$ for a local scheduler on $a \in \mathbf{A}$ by distributing the unused time of low-speed tasks over the more demanding ones.

By consecutively applying algorithm 5.4 to all targets, $\mathbf{s}$ is reduced to a feasible speed vector, where no processor has a load higher than 1. It is still an underestimation, though, since the limiting due to one processor may have an impact on the possible speeds of other processes on other processors.

We have applied a heuristic of reinitializing the speeds of all processes that are not running on a maximally loaded processor, and reiterating. This procedure converges because there is always at least one maximally loaded processor due to algorithm 5.4. The basis of the heuristic is that the speeds of processes running on a maximally loaded processor cannot be increased, while others may still be optimized.

$$WORK = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{ccccc} \textit{Sensor} & S \to A & \textit{ProcA} & A \to B & \textit{ProcB} \end{array} \\ \begin{pmatrix} 1 & 1 & 1 & & \\ & & 3 & 1 & 6 \\ & & & & 1 \\ & & & & 1 \end{pmatrix} \end{array} \qquad DEP = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccc} 0 & 1 & 2 & 3 \end{array} \\ \begin{pmatrix} & & & \\ 1 & & & \\ & 1 & & \\ & & 1 & \end{pmatrix} \end{array}$$

Figure 5.8: HDG for the edge detection DTIG of figure 5.7

Algorithm 5.4 Distributing the CPU time of a target $a \in \mathbf{A}$ over the tasks running on it.

```
Find tasks running on this target
```
$\mathbf{H}_a \leftarrow \{h \in \mathbf{H} | work_{h,a} > 0 \ \wedge \ \nexists h' \ dep_{h,h'} = 1\}$

$store = 1$
$members = \|\mathbf{H}_a\|$

```
Sort tasks ascending by their maximally usable fraction on this
target
```
for $i \leftarrow 1 \ldots \|\mathbf{H}_a\|$
  $h \leftarrow sortascending(\mathbf{H}_a, f_a)$

  ```
  Calculate the actual fraction
  Note that setting f_a(h) implies an s_h because of eq. 5.6
  ```
  $f_a(h) \leftarrow min(f_a(h), \frac{store}{members})$

  ```
  Update for next task
  ```
  $store \leftarrow store - f_a(h)$
  $members \leftarrow members - 1$

After the hypertasks' speeds have been determined, the model advances the time to the point when the first hypertask finishes, that is

$$\Delta t = \min_{h \in \mathbf{H}} \frac{1}{s_h}. \tag{5.8}$$

The work of all hypertasks is decreased by $f_a(h)\Delta t$, and dependencies of completed hypertasks are removed. The algorithm resumes with the new workloads and available tasks, and iterates until all tasks are completed. The predicted execution time is then the total amount of time that was advanced.

5.5 Buffer management

The stream task graph closely resembles an acyclic Kahn process network (KPN, [76]). In the Kahn model of computation, a number of processes are connected via unbounded FIFO channels. Reading from such a channel is *blocking*, while writing is *non-blocking*. A process may not test for the availability of input on a certain channel, and must be deterministic. These conditions ensure that the output of the network is independent of the schedule.

For practical implementation, we restrict this model to *bounded* FIFOs, and blocking writes. If the buffer sizes are not chosen correctly, this may introduce artificial deadlock. The most important task of the dispatcher is therefore to determine the sizes of the buffers that the remote functions use to communicate.

Another source of deadlock, unrelated to the KPN model, is the inability to schedule a certain operation due to resource conflicts. In this case we need to execute the *partial* process network that was constructed before the resource conflict

occurred, and wait until an operation completes and the resources are freed. This makes it necessary to buffer the outputs of the partial process network.

5.5.1 Buffers for data flow

An upper bound on the buffer space needed for deadlock-free execution is the stream size found during trace generation, because that means no stream write can block the program. This is impracticable as it requires too much memory.

In the Kahn model of computation, the lower bound on the buffer size is in general undecidable. We therefore use a technique by Parks [101], where the buffer sizes are determined dynamically. Since it is too expensive to do this at run-time, the maximum size of each buffer is recorded during a simulation run, and stored for use at run-time.

The technique works as follows:

1. Set the buffer sizes to some initial value
2. Run the program until deadlock occurs
3. Increase the size of the smallest full buffer
4. Continue with step 2.

This guarantees that the KPN runs in bounded memory *if the original unbounded KPN could run in bounded memory.* This condition holds if the skeletons comply with the application specification as described in section 5.3.2.

5.5.2 Buffer spilling

It may be impossible to schedule an STG because of a resource conflict. A typical case is trying to run two capture operations at the same time. In the background subtraction program 5.4, a background image `bkg` is subtracted from a foreground `img` and displayed. Then `img` becomes the background, and this is looped in an infinite cycle.

Program 5.4 Background subtraction code highlighting a sensor resource conflict.

```
capture(bkg);

while (1)
{
  capture(img);

  subtract(bkg, img, subt);
  display(subt);

  bkg = img;
}
```

If the architecture model contains one camera target providing one sensor resource, and the `capture` operation requires such a sensor resource, the second `capture` cannot be mapped. Further execution of the stream program must wait until the first `capture` operation completes and the sensor resource is freed. However, during this time the `bkg` stream will not be read, causing the buffer to be expanded to frame size.

As such frame buffers are often too large for resource-constrained embedded processors, they are *spilled* to a processor or dedicated memory with more space. All unconnected output streams are spilled in this way if an STG cannot be mapped.

5.6 Results

To evaluate our approach, we present results on the overhead associated with a locally scheduled, preemptive run-time environment, as well as the accuracy of our performance prediction and the performance of the resulting mappings. We will use four applications:

- *embarrass*, a synthetic embarrassingly parallel application with high arithmetic intensity (high computation-to-communication ratio) and few dependencies. Effectively a number of independent tasks.
- *stereo*, a correlation-based dense stereo vision algorithm using block matching, with 16 disparity levels and a 5x5 window as block size [127].
- *ball*, a Hough-based ball detection application that uses edge orientations to optimize the Hough transform [32].
- *SIFT*, implementing the keypoint detection part of the Scale Invariant Feature Transform described in [89]. It is based on detecting extrema in a difference-of-Gaussian scale-space.

5.6.1 Preemptive backend

The main difference between our streaming run-time environment and a normal image processing application is that the various operations are executed concurrently using a local scheduler. This increases task parallelism and reduces memory usage because the operations are pipelined. However, it introduces buffer interaction and context switching overheads.

In figure 5.9, this overhead is plotted relative to the frame-by-frame processing time. The overhead is larger for applications which deal with larger or more streams (such as the *stereo* and *SIFT* algorithms). The ball following application even experiences a speedup, because the pipelined processing increases cache locality.

Figure 5.10 shows the speedup obtained from task parallel execution on a 4-way symmetric multiprocessor using the preemptive backend. This indicates the maximum possible speedup which can be expected using our run-time system, since distributed-memory execution will only introduce more overhead. The speedup is

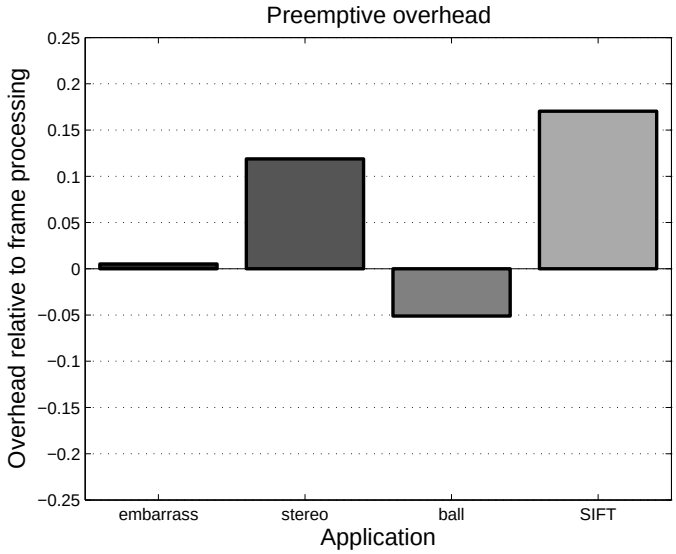


Figure 5.9: Overhead relative to sequential frame-by-frame processing on an AMD Opteron 250

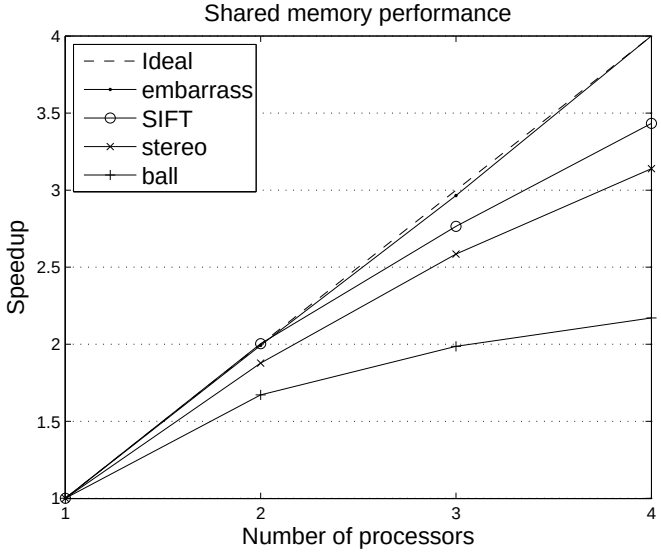


Figure 5.10: Symmetric multiprocessing speedups, indicating the inherent parallelism of our four applications. The results were gathered on a dual AMD Opteron 280.

in general quite low, indicating that in these applications, task parallelism should not be the main source of parallelism.

5.6.2 Performance prediction

As the mapping is based around a prediction of the resulting performance of a particular mapping, it is important that this prediction is accurate. We compare our method with three cases in which a particular aspect of our model is removed:

- No task dependency, or *TIG-only*, considering all tasks to be simultaneously executable. Often used for tightly coupled tasks such as in data parallelism. CREMA [115] is an example mapping algorithm for this domain.
- No task interaction, or *TDG-only*. Many scheduling algorithms, such as HEFT [124], ignore task interaction, and assume a task can only execute when all data has been received.
- No communication cost. This corresponds to scheduling algorithms for coarse-grained tasks such as PTS [108].

The relative prediction errors are plotted in figure 5.11[1]. It shows that the most important aspect is modeling communication, the absence of which leads to large prediction errors of the makespan. For the other aspects, the prediction error depends on the kind of application: *ball* (with the Hough transform having a bulk input) needs task dependency modeling, while *stereo* does not. DTIG performance prediction averages around 10% prediction error. Because TIG-only prediction is much faster, it can be advantageous not to model dependencies in situations where it achieves the same accuracy.

5.6.3 Mapping

A better performance prediction can be expected to lead to a better mapping. This is evident in figure 5.12, where unmodeled transports or task interactions yield significantly worse mappings. Again, the benefit of task dependency modeling depends on the application. Only in the *ball* application does it provide a significant improvement.

Note that it is possible that a bad prediction leads to a good mapping. The list scheduling algorithm described in section 5.3 is a greedy heuristic that is sensitive to small differences in the prediction. This is why the TIG prediction for the *SIFT* application in figure 5.12 yields a better mapping (although insignificantly so) than the DTIG prediction.

The distributed-memory speedup graph in figure 5.13 shows little difference with the shared-memory speedup in figure 5.10, indicating that computation and communication are well-overlapping. Only the ball detection algorithm suffers from the additional communication times, since its arithmetic intensity is low (it processes few operations per pixel). Beyond four processors parallelism starts

[1] Due to a problem with MPI threading, we are using 100Mbit Fast Ethernet interconnect in all DAS2-TUD measurements.

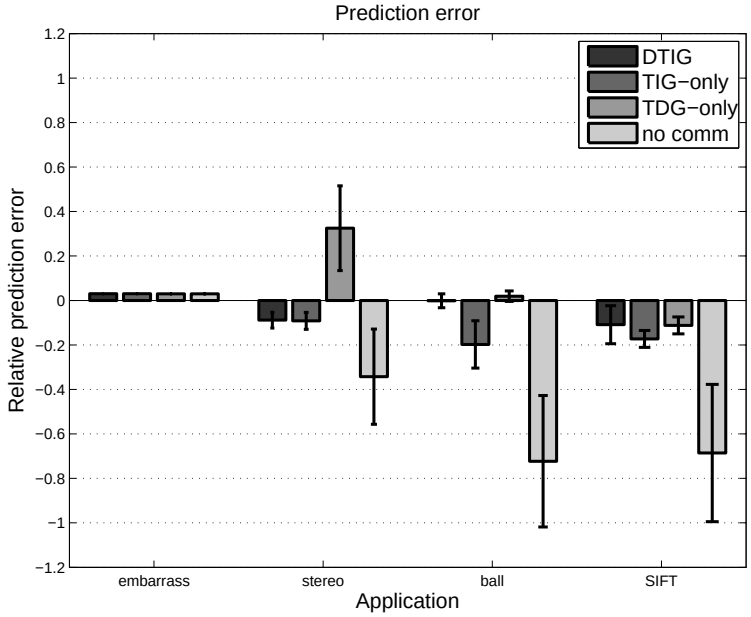


Figure 5.11: Relative performance prediction error ($\frac{t_{pred}-t_{meas}}{t_{meas}}$) as an average of mappings to 1-8 nodes of DAS2-TUD. A negative value means the prediction underestimated the makespan.

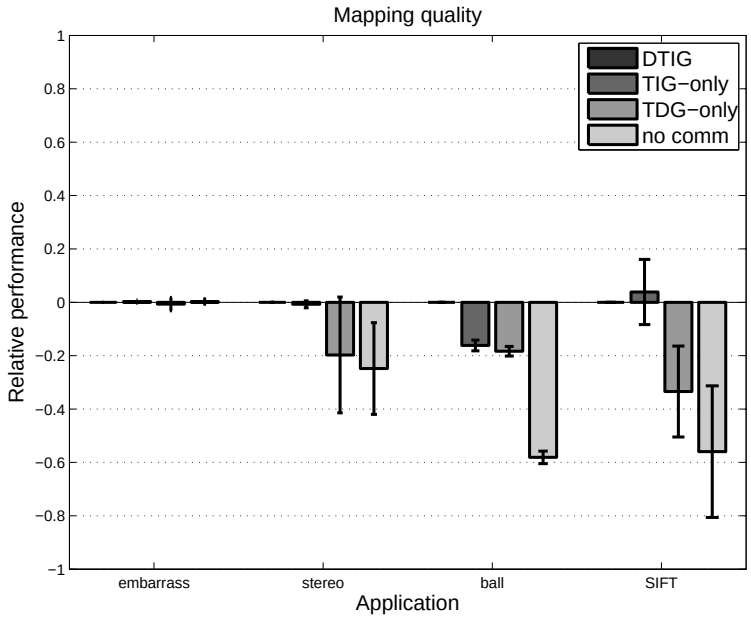


Figure 5.12: Relative performance difference as an average of mappings to 1-8 nodes of DAS2-TUD. The performance is relative to DTIG mapping.

to tail off, mainly because the applications do not possess vast amounts of task parallelism. Data parallelism, as exploited in chapter 4, will have to provide the best part of the speedup.

5.7 Discussion

We have presented a programming method and run-time environment for stream programming using future-based asynchronous RPC. Futures allow us to imperatively construct a task graph, and making dynamic stream reconfiguration automatic. Task parallelism is continued over such reconfigurations by running those decisions asynchronously as well. The task graph itself resembles a bounded FIFO Kahn process network, which can be mapped to arbitrary heterogeneous architectures with resource constraints.

The FIFO interactions introduce an overhead of around 10%, depending on the application. The achieved task parallel speedup is adequate for around three to six processors for typical applications, which is approximately the amount of processors we are aiming for in embedded systems. Larger speedups must be achieved using data parallelism, with the task parallel environment as a support for heterogeneous systems.

Futures-based RPC combined with algorithmic skeletons has previously been applied to grid computing, in [6]. However, they do not model communications and are as such limited to coarse-grained tasks. Combined TIG and TDG modeling has also previously been studied [110]. They report similar speedups, although their communication model does not include contention. This limits the architectures to which their approach is applicable.

As opposed to other work [108, 119], we do not consider the trade-off between data and task parallel processing while mapping; data parallelism is only exploited within a single target of the architecture model. The main reason is to avoid the data reorganization overhead and additional mapping complexity. We believe that a static trade-off (such as by the use of data parallel processors or static aggregations of sequential processors) provides sufficient performance. If desired, the trade-off could be made part of the design space exploration, described in the next chapter.

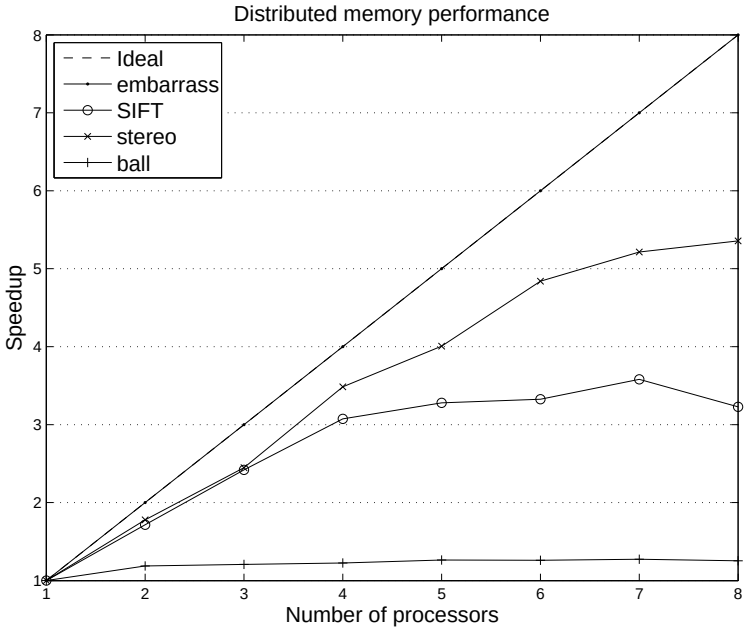


Figure 5.13: Distributed-memory speedups, as measured in DAS2-TUD.

Chapter 6

Exploring the SmartCam design space

Design space exploration [61] is the guided iteration over an architectural design space in order to optimize an objective function such as performance, power consumption or area. If this objective function is measured against a specific application or application domain, the result is an application or domain-specific architecture. Usually, multiple objective functions are optimized at the same time, resulting in a surface of optimal tradeoffs, called a *Pareto front*. The designer can then choose one of those tradeoff points as the final architecture.

In the previous chapters we have detailed which individual architectural components are suitable for embedded image processing, and how these can be programmed in an architecture-independent manner. Apart from ease of programming, the main reason we require an architecture-independent program is that it makes it possible to automatically change the architecture without requiring a rewrite. In turn, this enables a fully *automatic* design space exploration without any programmer intervention.

Figure 6.1 illustrates the basic flow of our design space exploration framework. An application is compiled using the tools described in chapters 4 and 5. This compilation uses an *architecture model* in order to instantiate the correct skeletons. The result is an executable (or set of executables) which is simulated to provide measurements such as performance and energy usage. These measurements are then used to instantiate an *architecture template* to a new architecture model, according to some search heuristic.

The architecture template is used to limit the design space, both because of efficiency concerns and because we have skeletons only for a specific set of processors. The template is described in section 6.1, while an environment for fast simulations is presented in sections 6.2 and 6.3. The instantiation heuristic itself is described in section 6.4. Finally, sections 6.5 and 6.6 present results and discuss them.

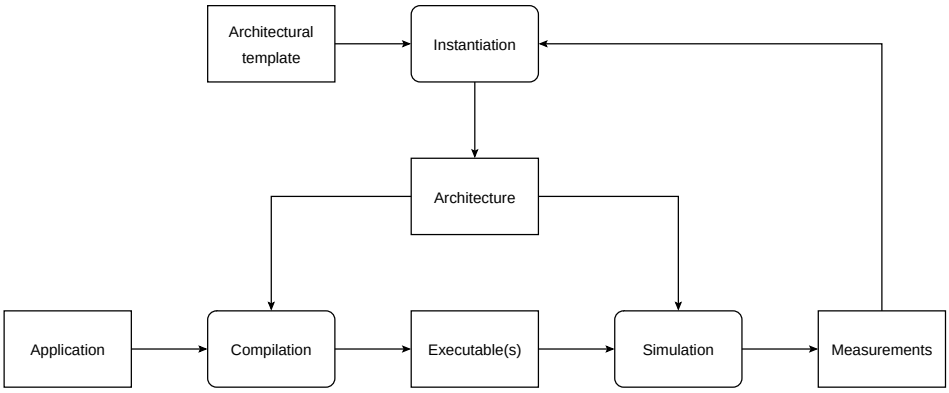


Figure 6.1: Flow of the design space exploration. The architecture model (used for both compilation and simulation) is iteratively adjusted based on the measurements.

6.1 Architecture template

An architecture template puts constraints on the legal architectures that the design space exploration may visit. An unconstrained exploration would be too expensive in terms of the time taken to reach a viable solution. Furthermore, unconstrained exploration of a processor's instruction set architecture requires a flexible retargetable compiler, which we do not have. The architecture template therefore constrains which processor architectures may be chosen (in a few variations), and how they may be interconnected. We have chosen architectures which make sense from an embedded image processing point of view (see section 2.2).

6.1.1 Processor types

As described in chapter 2, no single processor type can satisfy all our goals regarding performance, power consumption, and cost. We therefore include RISC, VLIW and SIMD processors in our template. More specifically, as fixed cores we include the TriMedia TM1100 5-issue VLIW and MIPS 24Kc RISC. We also include the XETAL SIMD processor, with 160 up to 640 processing elements (scaled from 320). The processors' properties are summarized in table 6.1.

Processor	**Frequency**	**Area**	**Power**
TriMedia	166 MHz	32 mm^2	1.1 W
MIPS 24Kc	240 MHz	7 mm^2	320 mW
XETAL	24 MHz	18 mm^2	50-200 mW

Table 6.1: Properties of the processors in our architecture template scaled to CMOS18.

6.1.2 Technology scaling

Not all the simulators we use and figures reported in literature refer to the same technology node. We have scaled all figures of area, power consumption, energy and frequency to CMOS18 using constant electric field scaling [19], meaning that the supply voltage is scaled as well. If L is the characteristic length of the technology node in which the figures were reported, then

$$\alpha = \frac{L}{0.18 \cdot 10^{-6}} \tag{6.1}$$

is the scaling factor. The die area scales with α^2, as the size is shrunk in both width and height. The supply voltage V scales with α, as does the capacity C. This means the maximum frequency f scales with α^{-1}. Furthermore, the scaled power P' and energy E' can be calculated from the original power P and energy E as follows:

$$P \propto CV^2 f \quad \Rightarrow \quad P' = P\alpha\alpha^2\alpha^{-1} = P\alpha^2 \qquad \text{and} \tag{6.2}$$

$$E \propto CV^2 \quad \Rightarrow \quad E' = E\alpha\alpha^2 \qquad = E\alpha^3. \tag{6.3}$$

6.1.3 Frequency scaling

We allow frequency scaling of the processors in our template, to achieve a more fine-grained objective space. Assuming we always use the minimum supply voltage required for a certain frequency, this has an effect on power consumption. Note that this is different from the technology scaling described in section 6.1.2, as we are changing the supply voltage while keeping the technology node constant.

We use the alpha power model from [112] (equation 6.4); for a desired scaled frequency f, we first calculate the scaled minimum supply voltage V, and then use this to find the scaled energy using equation 6.5. This assumes that dynamic power is dominant.

$$f \propto \frac{(V - V_t)^\alpha}{V} \tag{6.4}$$

$$E \propto V^2 \tag{6.5}$$

We use values which are nominal for CMOS18: $V = 1.8, V_t = 0.5, \alpha = 1.3$ ([59]). Frequency scaling was limited between 0.5 and 1.5 times the actual speed of the benchmarked processor, to avoid too many unmodeled effects. In particular, noise limits downscaling, while power density poses practical limits on scaling up.

6.1.4 Interconnect network

The SmartCam template considers three different interconnects: bus, ring, and fully connected. The entire interconnect has the same bandwidth, that is, all components are attached using the same type of ports. The area of the interconnect is modeled using MOSIS SCMOS layout rules for minimum-width METAL2 wires in CMOS18, at 100 MHz. We limit the bandwidth to either 100Mbps, 1Gbps or 10Gbps. The width of the interconnect is then

$$N_{wires} = \left\lceil \frac{bandwidth}{100 \cdot 10^6} \right\rceil . \tag{6.6}$$

The ring interconnect is segmented such that different segments may be in use at the same time. A processor can read from only one segment, and can only write to the next segment. As the size of the segment registers is insignificant compared to the wires themselves, we do not model them.

Full interconnect means that all processors may communicate simultaneously, but the total incoming or outgoing bandwidth of a processor may not exceed the network bandwidth. This is modeled as a full crossbar. Again, the multiplexers are not modeled, as their area is negligible in comparison to the wiring. If A_c is the area of a component $c \in \mathbf{C}$ (assumed to be square), then the wiring areas of the interconnect are calculated as follows:

$$A_w = 6\lambda \cdot \sum_{c}^{\mathbf{C}} \left(\sqrt{A_c} \right) \tag{6.7}$$

$$A_{bus} = A_{ring} = N_{wires} \cdot A_w \tag{6.8}$$

$$A_{fc} = N_{wires} \cdot A_w \cdot \| C \|, \tag{6.9}$$

where A_w is the area of a single wire traveling along all components (λ is related to the minimum feature size, and equals 90 nm for CMOS18). Assuming that the power dissipation from charging the wire capacitances C_w is dominant, the wire energy per bit transferred E_w is calculated as [137]

$$E_w = \frac{1}{2} C_w V^2 \qquad \text{(50\% duty cycle)} \tag{6.10}$$

$$C_w = 0.74\ fF/\mu m^2 \qquad \text{(0.2 } fF/\mu m \text{ at a width of } 3\lambda), \tag{6.11}$$

such that

$$E_{bus} = E_{fcline} = N_{wires} \cdot 3\lambda \cdot \sum_{c}^{\mathbf{C}} \left(\sqrt{A_c} \right) \cdot E_w \tag{6.12}$$

$$E_{ringsegment,c} = N_{wires} \cdot 3\lambda \cdot \sqrt{A_c} \cdot E_w. \tag{6.13}$$

E_{fcline} is the energy needed to use a single component-to-component path in the crossbar, and $E_{ringsegment,c}$ is the energy needed to use the ring segment that travels along component c.

6.1.5 Mapper

The architecture template also contains possible settings of the mapper. Instead of using the plain performance prediction described in section 5.4, the mapper can optimize for other variables. We use a weighted sum of the predicted makespan, energy, average processor utilization and average squared processor utilization. The last two are heuristics to promote locality and distribution respectively, and support a balanced mapping in the light of future, currently unknown tasks. Each weight is discretized to one of 0, 1, 2, 3, or 4.

6.2 Benchmarking

Figure 6.1 shows a multiprocessor executable being simulated on a multiprocessor simulator. While this is a viable and accurate approach, the effort required to integrate different processor simulators is prohibitive, as is the slow speed that may be expected from such a setup. Instead of this, we use processor simulators only to benchmark each operation in isolation, and combine them in a high-level multiprocessor simulation.

Figure 6.2 shows the details of our simulation environment. We simulate a single trace of an application. As the functional behavior is independent of the schedule (see section 5.5), we can create this trace on a normal workstation, and save all intermediate results[1]. These intermediate results are used to simulate each operation individually for each processor in an architecture. Such simulations can then be *cached* if the processor's microarchitecture does not change during design space exploration.

Finally, the benchmarked values are used to simulate the trace using a multiprocessor discrete event simulator, greatly speeding up the process. This is described in section 6.3.

6.2.1 Trace generation

An application trace is generated by executing the stream program on a workstation. Figure 6.3 shows an example trace of the ball following application that was used for evaluating the run-time system. The trace records the order of all operations, as well as their interconnections. These are used during application simulation, because the operation and stream identifiers are not deterministically generated (although the output remains deterministic).

The main reason for them not being fully deterministic are the encapsulated local functions described in section 5.1.2. Since they create new threads of ex-

[1]This means we ignore the effect a different architecture may have on the accuracy of the results.

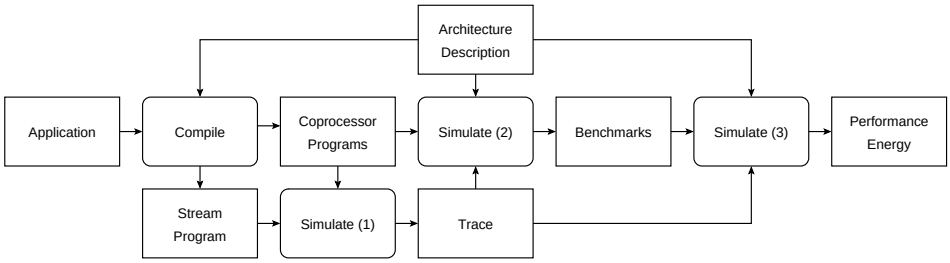


Figure 6.2: Simulation flow for the design space exploration. Simulation (1) runs on a workstation to generate a trace. (2) benchmarks each operation on each processor independently, using processor simulators. (3) uses these benchmarks and the trace to simulate the total multiprocessor architecture without actually executing the operations.

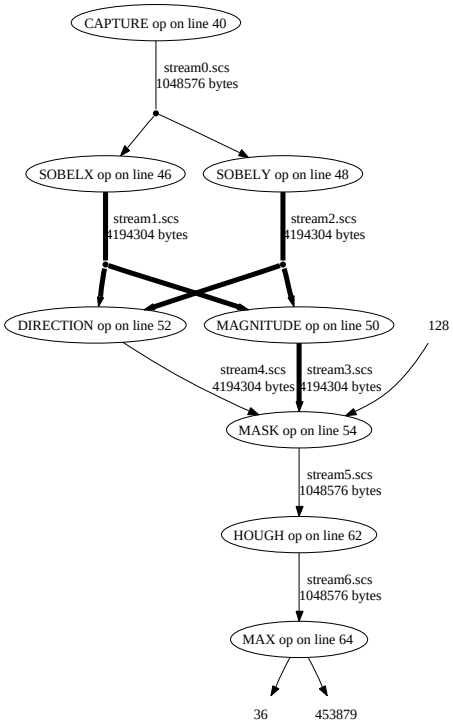


Figure 6.3: Trace for a ball following application. The round boxes denote operations (and the line of the program on which they occur). The edges are streams, annotated with the file in which they were saved and their lengths. The un-boxed numbers are non-stream arguments.

ecution, the order in which they call skeleton operations is arbitrary. However, the single-assignment semantics of futures (see section 5.2.2) guarantee that the stream interconnections are always the same.

The trace also captures the contents of all streams, and the values of all non-stream skeleton arguments; these are necessary to benchmark the individual operations. The latter are also needed to reconstruct the same dynamic branch decisions in the simulated application as in the original run. Note that this does not cover dynamic branch decisions originating from indeterministic behavior of the main application program itself, for example because it accesses additional operating system calls, or contains memory access errors.

6.2.2 Operation simulation

As mentioned before, all operations are benchmarked on each processor independently, assuming all data is available and the output buffers are large enough to contain the entire output. While this process is automated on workstations, we have not currently implemented the interfaces to the various processor simulators needed for automated benchmarking of the embedded processors. These are therefore conducted only semi-automatically.

The benchmarks also provide the amount of energy taken for the execution of an operation. This is assumed to be linear, so that all energies added up (including those of stream transfers over the interconnect) give the total energy taken for the entire application.

6.2.3 Cache effects

On processors that use cache memory, such as RISCs and VLIWs, the method of simulating individual operations and later combining the benchmarks is problematic. First, while the time it takes to switch contexts between operations is constant in the absence of caches, cache trashing makes this indeterministic, but always larger than the measured time. Second, the benchmarks are taken with a cold cache, while pipelining operations in the correct order results in a hot cache, improving performance.

We will not model these effects, assuming they are part of the general measurement uncertainty. In section 6.5.1 we present results on the achieved accuracy.

6.3 Application simulation

The final application simulation is performed by a high-level discrete event simulator. The simulator works by intercepting all communication calls made by the stream program's run-time system. Recalling figure 5.3 on page 86, it replaces the communication layer, leaving all other systems intact. The simulator can therefore simulate any application that may be expressed using the run-time system.

The simulator matches each dispatched operation to the trace, using its type and input values as keys. As mentioned before, streams are not matched by stream identifier, but only by connection: the stream should be generated by the correct operation. Once the correct operation has been located in the trace, the

benchmarks for that operation are consulted and used to determine the appropriate delays and power dissipation. After the operation completes, its outputs are returned to the gatherer.

The simulator only knows the total time it takes for an operation to process an image, and not how the processing time is distributed over the data that is read or generated. As in the model used by the mapper to predict the performance (see section 5.3), we distinguish between *regular* and *bulk* streams. A bulk stream is read in its entirety before the processing starts (or written out only after processing ends) while a regular stream is read during processing.

The main difference, then, between the performance prediction and the performance simulation is that the simulation takes context switching into account and works on an entire application, while the performance model assumes infinitesimal time slices and only works on partial process networks without dynamic branches.

6.3.1 Network model

The simulator uses an architecture model for simulating the multiprocessor system; the same model that is used by the mapper to predict the performance. The processors in this model are defined by the architecture template. Although some statistical information is known about these targets, such as chip area, the time it takes to execute different operations, etc, they are essentially black boxes.

Not so for the network model. There is no one "ring" network, because it is different depending on the number of processors which are attached to it. The network model is generated from the architecture template by creating a series of connected targets which together display the appropriate behavior.

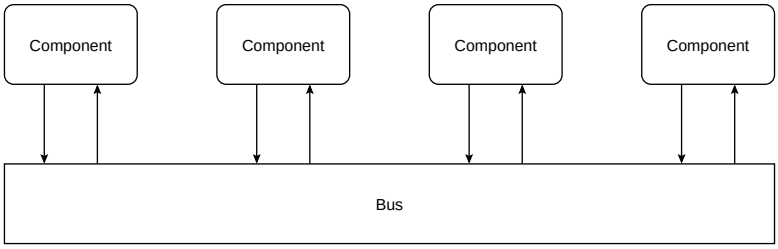
Figure 6.4 shows instantiations of the three network models for four components (processors). For the ring network in figure 6.4(b), a ring node is generated for each processor. The processor writes to this node, and reads from the previous node. The nodes are connected as well, in a unidirectional ring; each node has a certain bandwidth, and induces a delay. Longer transports therefore have a higher latency.

The fully connected case in figure 6.4(c) models a crossbar with multiplexers on the input. For each component an input and output node is created. All outputs are then connected to all inputs. Note that the bandwidth of the input nodes is the same as the bandwidth of the output nodes; it is not possible to receive multiple maximum-bandwidth streams.

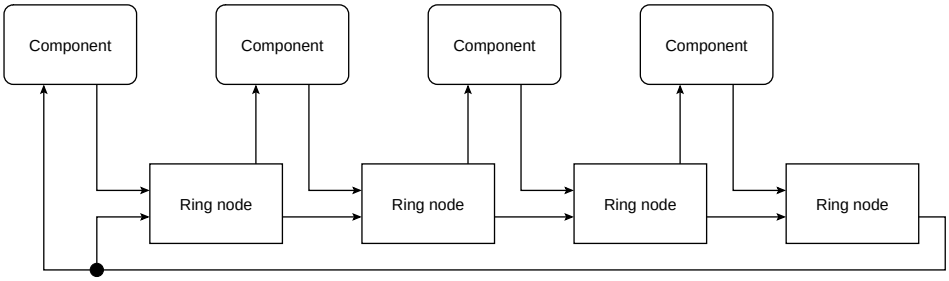
6.3.2 Discrete event simulation

We have created a light-weight discrete event simulator to model a multiprocessor network. All targets in the network are modeled as processors with cooperatively multitasking round-robin schedulers. Network nodes are processors which merely execute the "copy" operation. As such, bandwidth sharing is simulated by multitasking.

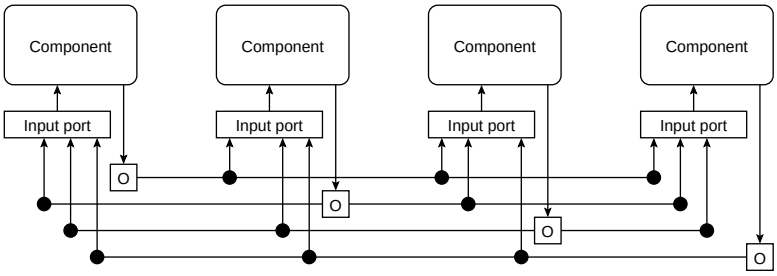
At the most basic level, all the simulator does is to maintain a global event queue, and provide a means to wait for and signal events. An event contains the time at which it is to occur, and the *process* it must run. Each process has its own



(a) Bus interconnect



(b) Ring interconnect



(c) Fully connected

Figure 6.4: Interconnection types

stack, and an event switches to this stack, resuming the process where it left off. There is also a `delay` function which adds an event to wake the process up after a certain amount of cycles.

However, this does not automatically model the contention of processes over the cycles of a processor, where only one process can run at the same time. Such software processes are called *threads*, and they are registered to a *device* which implements the scheduler. In this case, an event resumes the device instead of the process, allowing it to make scheduling decisions. Threads introduce a difference between `delay`, which occupies the processor, and `sleep`, which does not. Figure 6.5 shows a sequence diagram for a thread calling the `delay` function.

By creating more than one device, we can model multiple processors. They are synchronized because they share the same global event queue. This also means that communication is easily modeled by writing into another device's receive-queue and signalling an event which will wake up its reception thread. The simulator replaces the communications layer of our run-time system and relates the messages to a number of such devices.

By replacing only the communications layer, we can be sure that the rest of the system behaves exactly the same as in reality. Since the run-time system itself is multi-threaded, we provide an implementation of the `pthread` library based on this simulator. This implementation creates threads on the device of the currently running process, and also redirects the creation of and interaction with locking constructs such as mutexes and conditions to this device. The device then inserts the appropriate events into the global event queue.

Note that this is not a processor simulator. Time does not advance unless a process explicitly delays or sleeps for a certain amount of cycles. This means that while any program making use of the `pthread` library may be simulated, we do not get any valuable information about the execution time unless it periodically delays a realistic amount of time in order to model execution. In our case, these execution times are provided by the benchmarking step. Energy consumption is simulated using the `dissipate` function, which is typically called after each `delay`.

6.4 Pareto optimization

Using the area and energy model from the architecture template, and the performance figures obtained by benchmarking and high-level simulation, we wish to find the surface of optimal tradeoffs, called the *Pareto front*. A tradeoff point is called Pareto-optimal if there are no other design points which are better in at least one objective function, while not being worse in any other, i.e. it is not *dominated*.

Definition 6.1 (Pareto front) *Consider an objective space Y ($Y \subseteq \mathbb{R}^M$) which is to be minimized. A point $y \in Y$* dominates *a point $y' \in Y$ iff $\forall_i : y_i \leq y'_i \wedge \exists_i : y_i < y'_i$. The* Pareto-optimal set*, or* Pareto front *of a set of points are those points which are not dominated by any other points in the set: $P(S) = \{s | \nexists_{s' \in S} : s' \text{ dominates } s\}$.*

In general, we can only find $P(S)$ if we know S. However, we do not know our objective space Y (consisting of measurements), only a decision space X of

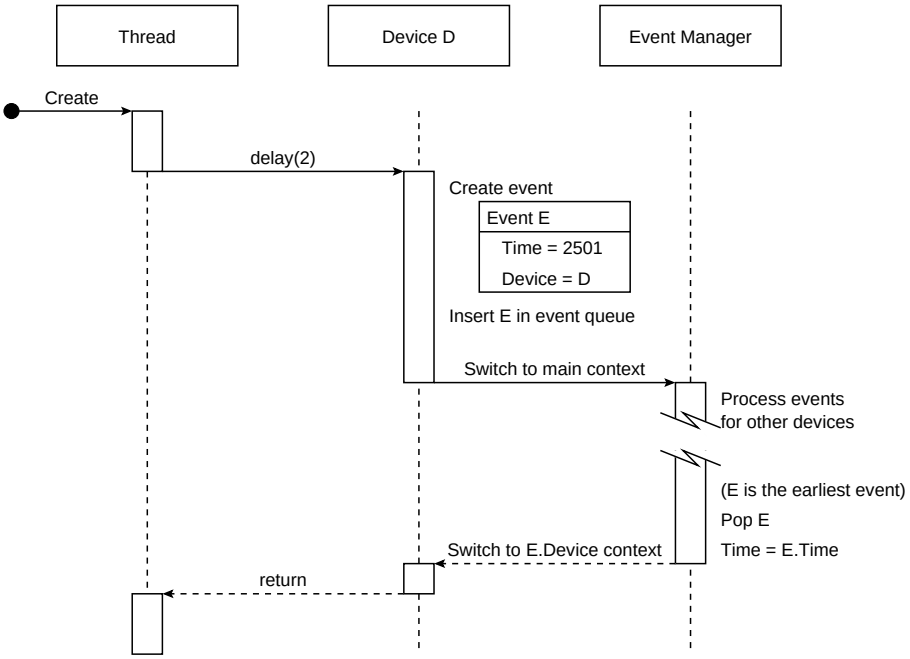


Figure 6.5: Sequence diagram for a waiting thread. The thread calls the `delay` function, which causes its device to add a wakeup event to the event queue, and switch context to the event manager. Once the event fires, the device is resumed, and `delay` returns.

architectures. This decision space is too large to evaluate completely. We therefore search only a part of it, and calculate the Pareto front of the visited elements.

6.4.1 Single-objective strategies

There are many ways in which to guide a multi-objective search [37]. A common one is to put constraints on all but one objective function, and optimize the remaining function using one of many single objective optimization routines, such as (iterative) hill climbing, simulated annealing, etc. For example, we might wish to find the cheapest architecture (in terms of area) which executes the program at a minimum of 10 frames per second, requiring no more than 100 mJ per frame.

$$\|y\| = \begin{cases} Inf & \text{iff } y_{time} > 0.1s \vee y_{energy} > 0.1J \\ y_{area} & \text{otherwise} \end{cases} \tag{6.14}$$

The most obvious drawback to this method is how to set the constraints. It might be possible that with a negligible performance loss, a much cheaper architecture is possible. Such a desirable solution would not be found because of the (partly arbitrary) constraint. Figure 6.6(a) shows this situation.

Another approach is to create a one-dimensional objective space by weighing the vectors:

$$\|y\| = w^T y. \tag{6.15}$$

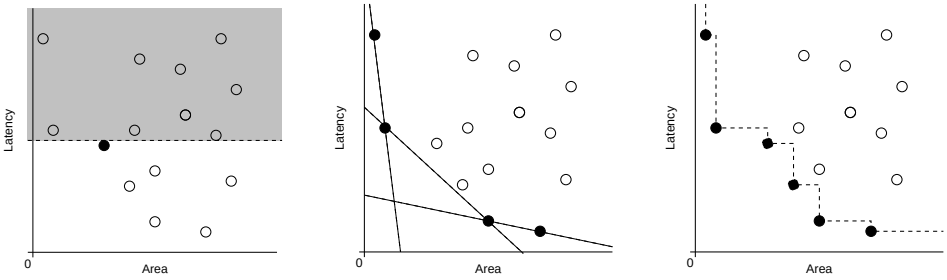
w can then be varied, so that we find multiple optimal tradeoffs. However, this limits us to a *convex* Pareto front, since we are only finding a single solution for each w; figure 6.6(b) illustrates this. Of course, we should remember all visited points, but this does not diminish the fact that we are still *optimizing* for only one solution for each weight.

6.4.2 Multi-objective strategies

In order to find a proper approximation to the complete Pareto front, we need to use a true multi-objective search strategy. Such a strategy tries to find an accurate and well-spaced approximation to the real Pareto set. Accurate, because we wish the set that is found to be as close as possible to the real Pareto front, and well-spaced because we are interested in a large surface area, not points clustered in one part of the front. Figure 6.6(c) shows the true Pareto front of a set of (2D) points.

Population-based search heuristics, such as genetic algorithms [58], are naturally suited to this domain. Since they already keep a population of solutions, we only need to make sure that this population is moved towards (approximating) a Pareto front. The Strength Pareto Evolutionary Algorithm 2 (SPEA2, [141]) is one such algorithm.

SPEA2 is an archiving, elitist genetic algorithm with strength and density-based fitness assessment. *Archiving* means that a second population (the archive) is kept during optimization, which contains the current best individuals. The



(a) Constrained single-objective. Left of the found solution is a point which is slightly slower but much cheaper.

(b) Tradeoff iteration. The dotted lines show the optimization directions using different weights, resulting in a convex set.

(c) True Pareto front, showing all non-dominated points. The dotted lines enclose the *dominated area*

Figure 6.6: Search strategies. The circles show all elements of the objective space; filled circles are those points found (in the limit) by a particular strategy.

individuals in this archive are not selected by any stochastic means, but by simple sorting of the fitness values. The algorithm is *elitist* because the (stochastic) mating selection also uses this archive. Mating selection is implemented using binary deterministic tournament selection, and the selected individuals are varied using mutation and recombination. This process is summarized in equation 6.16

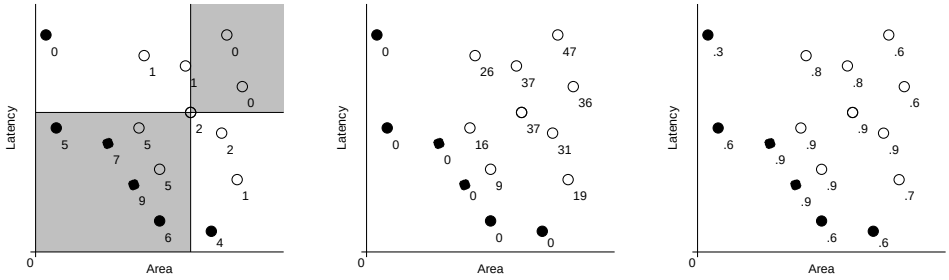
$$\overline{P_t} + P_t \xrightarrow[\text{(sorting)}]{\text{archiving}} \overline{P_{t+1}} \xrightarrow[\text{(tournament)}]{\text{selection}} M \xrightarrow[\text{(mut./rec.)}]{\text{variation}} P_{t+1}, \tag{6.16}$$

where P_t is the population at generation t, while $\overline{P_t}$ is the corresponding archive and M is the mating pool.

Fitness assessment needs to accomplish the twin goals of accuracy and spread. For the first, SPEA2 uses the combined *strength* of an individual's dominators, where the strength of a design point is the number of individuals it dominates. Thus, if fitness is minimized, non-dominated points are favored, followed by points whose dominators dominate few points. This is illustrated in figures 6.7(a)-(b).

Spread is ensured by adding an individual's reciprocal distance to its kth-nearest neighbor to the fitness value (a smaller distance, e.g. higher density, therefore leads to a worse fitness). Ties between points with the same dominators' strength are broken using this measure. This is shown in figure 6.7(c).

We have chosen to use SPEA2 because it performs comparable to other current approaches in evolutionary multi-objective optimization, and uses the PISA [17] platform-independent interface for search algorithms that is easily adaptable to different problem domains.



(a) The strength of a point is the number of points it dominates, i.e. the number of points in the upper right-hand shaded region

(b) The fitness is the combined strength of a point's dominators: the sum of the values in the lower left shaded region in figure 6.7(a)

(c) The density added to the fitness value is one over the distance to its kth-nearest neighbor. Here, $k = 1$.

Figure 6.7: Fitness assessment by SPEA2, based on dominators' strength, and density.

6.5 Results

There are two important aspects that need to be measured in order to verify our design space exploration. We must know whether our simulation accurately reflects real-world performance, and whether the solutions found by the search heuristic accurately approximate the real Pareto front. We also investigate the results of the exploration using two case studies.

6.5.1 Application simulation

Our discrete event simulation takes a highly abstract view of a multiprocessor system and application, which could lead to inaccuracies. We have therefore modeled the DAS2-TUD cluster described in section 2.2 with various numbers of processors, and compared the simulated with the measured performance for a number of applications.

Figure 6.8 shows the results. *stereo* and *SIFT* were already used in the previous chapter, while *deptest* and *AR* introduce dynamic branches into the application. *deptest* is a simple test consisting of pixel operations and one frame summation, dependent on which other pixel operations are executed. *AR* is a position estimation application for augmented reality, based loosely on [26]. It tries to measure the position of a printed pattern using Canny [34] edge detection and edge following, or, failing that, natural features using SIFT. The dynamic branch is introduced by the probability of a pattern being detected or not.

The simulation accuracy is around 10%, depending on the application. It is slightly better than the accuracy of the DTIG performance prediction presented in chapter 5, and retains the same accuracy in applications with dynamic branches. However, discrete event simulation can be a relatively slow process, and for applications without dynamic branches it is therefore appropriate to use the performance prediction during design space exploration.

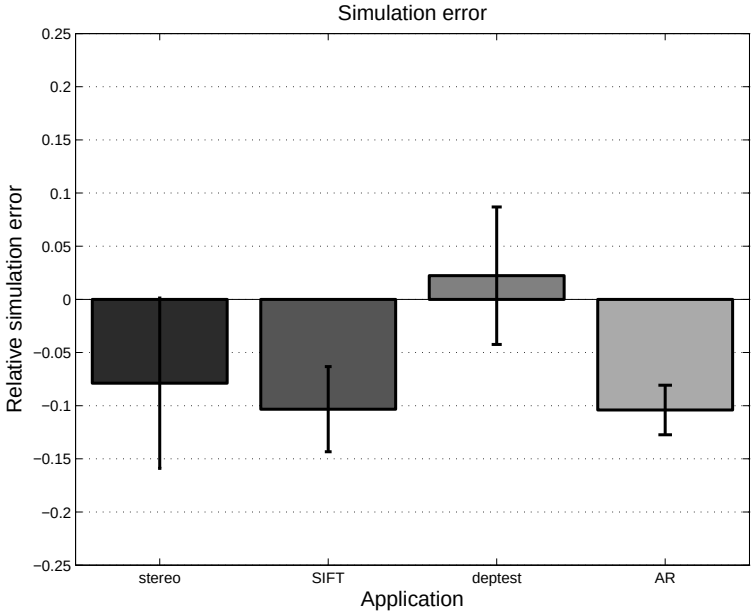


Figure 6.8: Relative simulation error as an average of mappings to 1-8 nodes of DAS2-TUD. The simulation error is relative to the actual makespan; a negative value means the simulation underestimated the makespan.

6.5.2 Convergence

We investigated the convergence of the SPEA2 heuristic to the optimal solution by running the exploration on a simple application and reduced architecture template, allowing a comparison with a brute-force approach. The application (*robocup*) is the front-end vision of a soccer-playing robot, detecting the positions of the ball, goals and other robots[2]. The architecture template was reduced by removing the scaled XeTaL processors and disabling frequency scaling.

Figure 6.9 plots the fraction of space dominated by the population as a function of the number of generations. This fraction is measured in a $(0.1s, 100mm^2, 0.1J)$ cube (normalized in each dimension), which is the part of the objective space that we are interested in. Convergence ends around generation 35 with an average dominated space of only 1% less than the optimal solution attained by a brute-force search.

6.5.3 Case study: Robocup

Our first case study is the *robocup* application used to test convergence. This is a simple application, consisting of few operations and without dynamic branches. It starts by color conversion and segmentation, and continues with run-length

[2]Application kindly provided by Harry Broers of the Philips RoboCup team and benchmarked by Hamed Fatemi of the Eindhoven University of Technology.

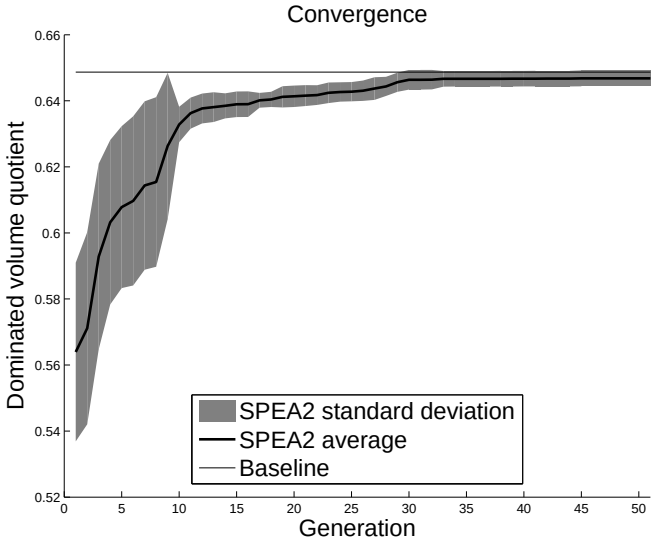


Figure 6.9: Convergence of SPEA2 for the *robocup* application. Archive and population size were 50. Average and standard deviation over 10 runs. The baseline was generated by brute-force search.

encoding, connected component analysis and blob analysis for the different objects it has to detect. Only the color conversion and segmentation can be executed on the XETAL processor.

In figure 6.10 we have plotted the dominated space of a typical SPEA2 run (filled space) versus that of the combination of 25 runs (black lines). Each corner is an architecture that dominates all architectutures in the block behind it. By plotting the combination of all such blocks, we visualize the dominated space. It can be seen that the difference between the filled blocks and unfilled ones is fairly minimal, indicating that one run provides a good approximation to the optimal Pareto front.

A few interesting points are indicated. Point A is the cheapest architecture that falls within the limits, consisting of two 24Kc processors connected over a 100Mbps bus. Point B swaps one 24Kc with a 160-PE XETAL running at half speed, while point C runs the 24Kc at 1.5 speed. From this progression, we can see that in this application the amount of data parallelism is such that a half-speed, 160-PE XETAL can keep up with a 1.5-speed 24Kc.

Points D-F form another progression, and consist of a TriMedia and 160-PE XETAL running at 0.5, 1.0 and 1.5 speed respectively, with corresponding increases in performance and power consumption. The TriMedia is faster than the 24Kc, and therefore requires a higher clocked XETAL to keep the balance. These architectures also switched to a ring interconnect, for increased bandwidth.

Finally, point G is one of the most expensive architectures found, with two TriMedias (one at normal speed, one at 1.5 speed) and a 640-PE XETAL. The interconnect is also upgraded, to a 1Gbps ring.

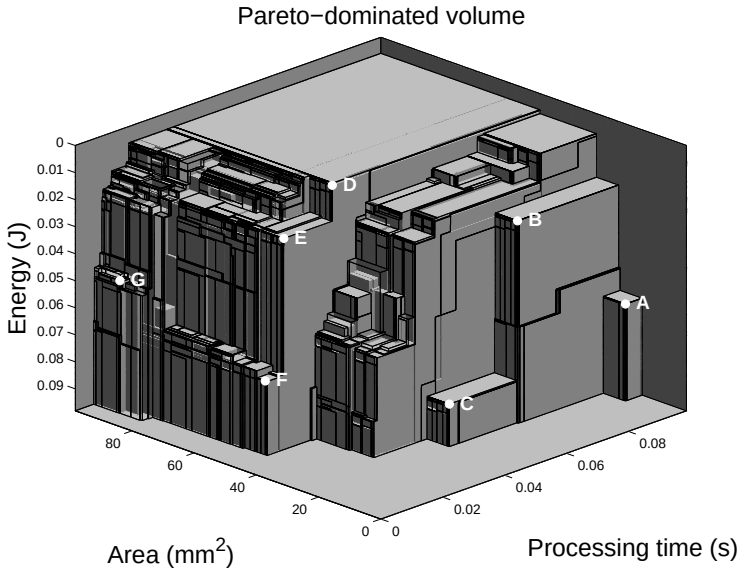


Figure 6.10: Dominated space of one SPEA2 run (100 generations) versus the combination of 25 runs for the *robocup* application. The performance measures are given for the processing of a single frame.

6.5.4 Case study: Augmented Reality

The AR position estimation application consists of two main parts: a Canny edge detection and SIFT keypoint extraction; whether the SIFT component is executed depends on the output of the edge detection. The edge detection consists of measuring the gradient magnitude, binarization using a threshold determined by a histogram, non-maximum suppression in the gradient direction, and thinning. We also calculate a "cornerness" value for all edge pixels, using the gradient structure tensor. In our simulations, we assume a SIFT step is needed every 10 frames, and therefore simulated 10 frames of processing, with a SIFT step at the first frame.

The resulting Pareto front is plotted in figure 6.11. The fastest architecture (running at 6.5 FPS) is point A with two TriMedias, two XeTaLs and three 24Kcs (all at 1.5 speed). A more reasonable architecture (point B, 5 FPS) uses only one TriMedia, one XeTaL, and two 24Kcs. It is also interesting to look at point C, which differs from A only in the mapper settings by emphasizing energy consumption over execution time. It is 0.1% slower while consuming 4.6% less energy.

To verify that we have indeed found an application-specific Pareto front, we have re-simulated the AR application on the Pareto points found during the *robocup* case study. We then compared the fronts by calculating the quotient of their respective exclusively dominated spaces with respect to reference ($10s$, $150mm^2$, $20J$) (see figure 6.12 for an explanation and definition). In this comparison, the space exclusively dominated by the real AR search was 3.1 times that of the *robocup* front, indicating that the front is indeed application-specific.

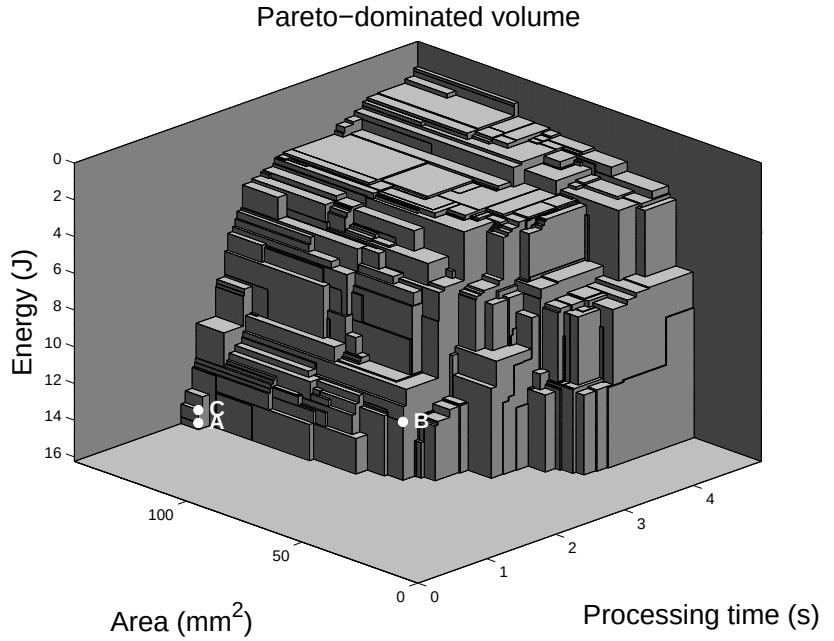


Figure 6.11: Dominated space of the AR application. The performance measures are given for the processing of ten frames.

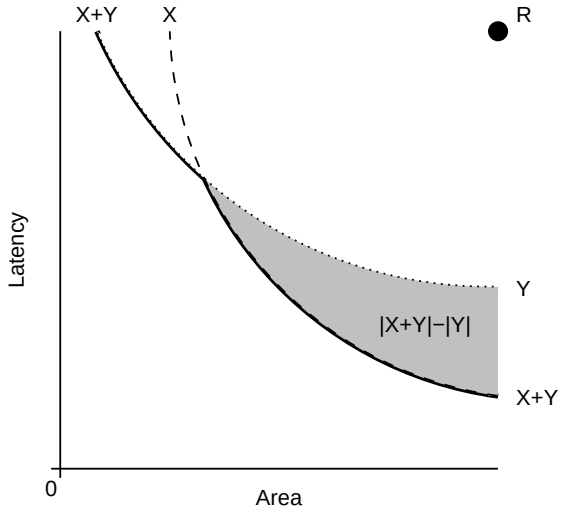


Figure 6.12: Comparing two Pareto fronts using the quotient of their respective exclusively dominated subspaces. The gray area is the subspace exclusively dominated by front X, and is calculated as $|X+Y|-|Y|$ ($|X|$ is the fraction of space dominated by X, with respect to R). The quotient is therefore $\frac{|X+Y|-|Y|}{|X+Y|-|X|}$.

Figure 6.13 shows another output of the simulation, which is the average utilization of each processor. This information can be used to gain insight into the results and to support the designer in a manual exploration of the design space, potentially leading to a restructuring of his program. In this case, it indicates that the network bandwidth is more than sufficient, and that the application is dominated by operations which cannot execute on the XeTaL SIMD processor. Although the *AR* application would be dominated by filtering on a sequential processor, the speed of XeTaL compared to the sequential processors is so large that the situation reverses. Amdahl's law [8] applies, and the application is now dominated by its sequential part.

This suggests that it is worthwhile to either investigate an implementation that uses more restrictive skeletons, or to include a more general SIMD processor, such as an IMAP-CE. In both cases, this will reduce the load on the sequential processors.

6.6 Discussion

We have presented an architecture simulation platform for heterogeneous embedded multiprocessors, based on discrete event trace simulation. The simulator was shown to have an accuracy of around 10% in typical applications. Based on this simulation, we have created an automated design space exploration environment for skeletonized applications, which quickly converges to an adequate Pareto solution surface.

High-level discrete event simulation is a proven concept [62, 129], including during design space exploration [87, 104]. Our contribution is the automated creation of the event model from an application using trace generation and benchmarking, in which skeletons provide the necessary information about the operations. The user of the exploration environment therefore does not need to understand the model.

The design space exploration currently only varies the architecture and mapping. Taking this approach even further, an operation may be implemented using *multiple* algorithms. For example, and iterative erosion can be specified as a suc-

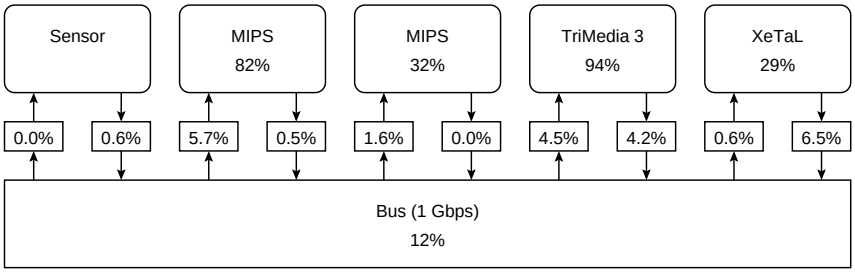


Figure 6.13: Processor utilization for the *AR* application on a network of two 24Kcs, one TriMedia and one XeTaL (point B in figure 6.11).

cession of local neighborhood operations, as a recursive neighborhood operation, or as a stack operation. If the user specifies all possibilities, the design space exploration could automatically determine the best implementation.

Chapter 7

Conclusions

Parallel heterogeneous multiprocessor systems are often shunned in embedded system design, not only because of their design complexity but because of the programming burden. Programs for such systems are *architecture-dependent*: the application developer needs architecture-specific knowledge to implement his algorithms, as each processor has its own characteristics and programming language. He will therefore often stick to the architectures he knows best instead of looking for the best one. This leads to suboptimal solutions, and costly redesign efforts if the chosen architecture later proves to be insufficient. We demonstrated that there is no single best architecture or programming language that can release us from this plight (chapter 2).

Our solution to this problem uses a programming model based on the concept of *architecture independence through algorithm dependence* (chapter 3). By limiting the expressiveness of a programming language to just those concepts needed to implement a given class of algorithms, it may be compiled to a variety of different (parallel) processor architectures. In particular, we limited the access patterns an algorithm may employ using *algorithmic skeletons*. Using different languages – or *interfaces* – to implement different algorithms is more natural than using architecture-dependent languages, because a programmer should be concerned with algorithms, not with processors. A language that is tailored to a specific class of algorithms will allow those algorithms to be expressed more naturally than using a general-purpose or processor-specific language.

We have designed a meta-programming system to implement translators for such *algorithm-specific languages* (ASLs). A translator for an ASL has three distinct tasks. First, it must implement the access pattern that is part of the skeleton, such as a row-major iteration over image pixels. Second, it must rewrite the algorithm-specific interface to that used by the run time system – for example, changing relative array indexing to absolute indices. Finally, it must translate the algorithm (also called the *kernel*) into a language that is understood by the target processor. PEPCI, a new meta-programming language and tool for implementing ASLs, facilitates these tasks by allowing the pattern to be implemented directly in the target language while still providing sophisticated code transformation tools (chapter 4). The operations resulting from the translation were shown to perform

within 20% of their handcrafted counterparts, in which the latter can be considered fairly optimal.

An application consists of multiple algorithms, each written in its own language dialect. In image processing applications, these algorithms often work in successive steps on a series of images. This suggests the view of an application as a directed graph of tasks working on streams of data. This is the view taken in *stream programming*, and we presented our implementation of that concept (based on *remote procedure call (RPC)*) (chapter 5). Using *futures*, RPC allows the application's task graph to be built under imperative program control, thereby allowing it to be data-dependent. Such data dependencies are important in applications that need to operate in dynamic environments, or in dynamic roles. An automated, dynamic mapping of the task graph to an *architecture model* maintains the architecture independence of the program. We demonstrated that this mapping creates an effective multiprocessor program, but that most of the benefits of using a heterogeneous system must be realized by parallelizing the operations themselves.

Finally, we used the architecture independence of programs written using our programming model to conduct an automated *design space exploration* of possible architectures, creating a *Pareto front* of optimal trade-offs between performance, area and power consumption (chapter 6). Each design point was simulated, using pre-benchmarked operations to speed up the process. The simulation has an accuracy of around 10%. We used *multi-objective evolutionary optimization* to approximate the Pareto front, and showed that this has good convergence and coverage. The practicality of the entire system of skeletonization, stream programming and design space exploration was demonstrated by implementing two case studies.

7.1 Discussion

The SMARTCAM framework tries to bridge the gap between the development of algorithms done by image processing researchers and their implementation in real (embedded) systems. Other systems have been proposed for the same purpose, many based on *visual programming* [82, 93]. They often provide a sophisticated graphical user interface to connect algorithm components, which can be part of a library or user-created. Our approach is not incompatible with such systems; on the contrary, the skeletons fill a crucial gap between user created and library components, and the task graphs used by our run-time system can be generated from the visual program.

We have chosen to use the C language as the basis of our framework because of its widespread use in embedded systems. C compilers are available for almost all processors, although they use different dialects. Using C as a basis therefore provides a certain amount of homogeneity. Yet image processing applications are often prototyped in higher-level languages, such as MATLAB. Although we feel that our C-based algorithm-specific languages provide enough of an abstraction from low-level C programming to be usable for image processing researchers, for strict *users* of image processing it would be interesting to investigate even higher levels of abstraction, and how they impact the compilation process.

Pseudo-dynamic meta-programming, that is the combination of a multi-level language with partial evaluation (section 4.3), is a promising meta-programming paradigm. By leveraging an existing language, we only need to add a minimum number of new constructs to allow meta-programming. Partial evaluation then specializes the meta-programming constructs, leaving only original language constructs. This approach is not limited to skeleton instantiation, but can be used in any situation where the meta-program does not depend on run-time values. However, it is most sensible when the compiler for the original language is not optimized enough, or can only compile a subset (in which case the meta-program provides a translation into that subset).

An important aspect of all programming systems is how they can be debugged. Introducing skeletons adds two additional language levels at which errors can occur: the skeleton itself, and the target language. Although many errors (such as type errors) are caught during skeleton instantiation, others will only become evident during actual execution. As the final program is quite different from the user-provided kernel, debugging becomes problematic. If the skeletons are correct we can at least debug the kernel using only one implementation, such as a sequential workstation backend, but the problem remains unresolved.

The real-world measurements on multiprocessor systems presented in chapter 5 were conducted on a computer cluster, not on an embedded system. The reason was that we could not construct arbitrary heterogeneous embedded systems and conduct the measurements on them. Of course, the SmartCam compiler and run-time system can be applied to such clusters as well. However, we do not currently exploit data parallelism between the processors, as this is accomplished by the SIMD processors in our architecture template. As most of the parallelism in image processing application is of the data parallel type, a proper SmartCam runtime for clusters will therefore split the cluster into a number of parallel *virtual* processors, exploiting both data and task parallelism. Our skeletons can support those virtual processor targets by compiling the kernel into an MPI program. In a similar way, we can target multi-core processors for data parallel execution by writing a threaded implementation of each skeleton.

We treated custom logic solutions such as FPGAs only cursorily, although they are increasingly popular in embedded applications. FPGAs can be supported in two ways: first, by providing parameterizable fixed-function library components, and second by instantiating the kernels into a language that can be *synthesized* into logic, such as Handel-C. The first option is often used because user programming of FPGAs is hard, and it can be readily incorporated into our system. Section 2.4.4 showed, however, that a kernel will have to be heavily transformed before it can be efficiently synthesized. It is therefore probably best to start with supporting a very limited set of restricted algorithms and expand from there. Restrictions on the expressiveness will ensure that the transformations can be kept simple.

Another possible platform that we have not elaborated on are smart camera networks [38, 81]. Our system has no inherent limitation to tightly coupled networks such as chip multiprocessors, printed circuit boards or computer clusters. Other systems are targetable by correctly modeling the bandwidth of the communication channels. However, since our framework views the entire network as a

single multiprocessor, this poses the question of reliable operation under component failure. As RPC systems routinely address the issue of reliability, they would again be a good starting point for further investigation.

Many improvements can still be made to the design space exploration. Most importantly, an integration of the various processor simulators would allow for a fully automated exploration from source program to target architecture. But more user direction could improve the system as well. We created a preliminary tool in which the user can graphically and interactively investigate the 3D Pareto surface and try alternative configurations. Possible options are to allow the user to provide an initial guess of the desired architecture, set constraints, and prune undesired branches during the search.

The patterns of computation introduced by algorithmic skeletons are a paradigm shift similar to the Reduced Instruction Set Computer (RISC, [35]) movement in the early 1980's. Instead of using processor architectures which can execute all patterns slowly (such as superscalars or VLIWs), we use architectures which support only a few patterns, but can execute those very efficiently. However, whereas the RISC concept was used to create general-purpose processors, our design space exploration automatically determines which patterns should be supported for a specific application or domain, and to what degree.

Glossary

Abstract syntax tree (AST) A finite, labeled, directed tree of terms that is the result of noting which productions were used while parsing a sentence in a (computer) language using a grammar, thereby providing an explicit description of its structure (syntax). Unlike other parse trees, abstract syntax trees only contain the applications of productions that impact the meaning of the sentence, p. 55.

Algorithm-specific language (ASL) A language (usually based on another language) created specifically to implement a particular class of algorithms. Similar to a skeleton in that the language often includes a predetermined iteration strategy. Our ASLs are implemented as compile-time meta-programs, p. 34.

Algorithmic skeleton See skeleton, p. 35.

Application specification Information about the operations that are used in an application, such as the intent and distribution of their stream arguments, and benchmarks on all targets in the relevant architecture model, p. 90.

Architecture model A graph and associated information about the available targets in a multiprocessor system, p. 89.

Architecture template A template for constructing architecture models. Used to constrain a design space exploration, p. 108.

Client A program, running on the master control processor, containing RPC calls, p. 81.

Dependent task interaction graph (DTIG) A mapping of a stream task graph to an architecture model. The edges between task vertices can be either *dependent* or *interacting*, thus combining a task dependency graph and task interaction graph, p. 92.

Design point A point in design space. That is, a particular instantiation of an architecture template, with associated objective values for a certain application, p. 116.

Discrete event simulation A way of system simulation to observe the dynamic behavior of a system. A DES model consists of *entities* able to process and post *events*, which are labeled by the time at which they must occur. The simulator iteratively advances time to the first event in a global *event queue* and executes the corresponding entity, p. 114.

Future A token, returned by an RPC stub, that can be used to access the result of the call, p. 84.

Grammar A set of rules, or *productions*, which describe the syntactical structure of a language. The productions relate words in the language to terms which can themselves be the input to further productions, p. 55.

Hypertask A combination of tasks, possibly spanning multiple targets, which can be regarded as a single task for the purposes of performance evaluation, p. 96.

Hypertask dependency graph (HDG) A task dependency graph in which the tasks are hypertasks, p. 96.

Instruction A primitive (arithmetic) operation executed by a processor, p. 9.

Kahn process network A model of computation where a number of concurrently running deterministic *processes* are connected by unbounded *buffers*. Reading from a buffer is blocking, while writing is nonblocking, and a process may not test for the availability of input. The output of a KPN is independent of the schedule, p. 99.

Kernel A function used to instantiate a skeleton into an operation. Usually, the kernel specifies only the way to process a single datum. Also, a mask used during image convolution, p. 35.

Language specialization An application of partial evaluation where a language is specialized to a subset of itself, because all constructs not in the subset are only used in static expressions, p. 66.

Master control processor (MCP) The target which executes the client, p. 131.

Meta-program A program that manipulates other programs. Examples are compilers and source-to-source translators, p. 53.

Meta-skeleton A way of restricting the expressiveness of a skeleton so that it can be reasoned about. Also, a meta-program which implements a transformation based on such an analysis, p. 73.

Operation A remote function running on an RPC server. Generally, operations are instantiated kernels which work on streams, but this is not required. For example, fixed-function operations are provided which capture an image, write it to mass storage, etc, p. 37.

Pareto front A surface of optimal tradeoff points in a multiobjective optimization problem. Each point cannot be improved in any aspect, unless another aspect is worsened, p. 116.

Partial evaluation A combination of aggressive optimization techniques used to execute as much of a program at compile-time as possible. Also called *program specialization*, because it specializes a program to the known inputs that are provided to the partial evaluator, p. 65.

Processing element (PE) A functional unit in a processor capable of executing instructions. The number of PEs in a processor determines the maximum amount of parallelism it can exploit, p. 9.

Remote procedure call (RPC) A way of starting operations on remote servers that mimics function call semantics, p. 81.

Residual program The parts of a program that could not be optimized away using partial evaluation, and that are output to the next stage of compilation, p. 56.

Rewriting Modifying a term through the successive application of rewrite rules. Mathematically the rules may be nondeterministic, but programming languages based on rewriting impose a (possibly user-defined) order, p. 61.

Server Any target which can execute one or more operations. Also, the program running on such a target, p. 81.

Skeleton A higher-order function specifying the iteration strategy with which to apply a kernel. Also, any other specification or implementation of such a strategy. See also algorithm-specific language, p. 35.

Source-to-source translation Using a meta-program to translate a program written in a high-level language into another program written in (another) high-level language. Often used to avoid the generation of assembly language when implementing a compiler, p. 37.

Stream A sequence of data elements of the same type, p. 38.

Stream programming The art of writing programs which consist of kernels operating on streams. Essentially a data-centric approach, similar to dataflow programming on a macro scale, suited to parallel distributed memory architectures, p. 38.

Stream task graph (STG) A directed acyclic graph interleaving tasks and streams. Created by future tracking, p. 91.

Target A processor or network node in an architecture model, p. 89.

Task A particular instance of an operation or data transport, p. 91.

Task dependency graph (TDG) A graph specifying the dependencies between tasks. A task cannot start until all the tasks on which it is dependent have finish execution, p. 93.

Task interaction graph (TIG) A graph specifying the interactions between tasks. All tasks execute simultaneously, p. 93.

Term A subtree in an abstract syntax tree or other parse tree. Its label is defined by the production in the grammar that produced it, and its children are the subterms that were the input to said production, p. 61.

Bibliography

[1] A.A. Abbo, R.P. Kleihorst, L.Sevat, P. Wielage, R. van Veen, M.J.R. op de Beeck, and A. van der Avoird. A low-power parallel processor IC for digital video cameras. In *Proc. 27th European Solid-State Circuits Conference, Villach, Austria*. IEEE Solid-State Circuits Society, Sept. 18–20 2001. p. 12.

[2] M. Achour, F. Betz, A. Dovgal, N. Lopes, P. Olson, G. Richter, D. Seguy, and J. Vrana. *PHP Manual*. PHP Documentation Group, May 2007. p. 55.

[3] D.A. Adams. *A computation model with data flow sequencing*. PhD thesis, Computer Science Department, Stanford University, 1968. No. CS-117. p. 43.

[4] G. Aigner, A. Diwan, D. Heine, M. Lam, D. Moore, B. Murphy, and C. Sapuntzakis. An overview of the SUIF2 compiler infrastructure. Technical report, Stanford University, 2000. p. 56.

[5] M. Aldinucci and M. Danelutto. Stream parallel skeleton optimization. In *Proceedings of the 11th IASTED International Conference on Parallel and Distributed Computing and Systems*. IASTED/ACTA press, 1999. p. 71.

[6] M. Aldinucci, M. Danelutto, and J. Dünnweber. Optimization techniques for implementing parallel skeletons in grid environments. In S. Gorlatch, editor, *4th International Workshop on Constructive Methods for Parallel Programming (CMPP'04)*, pages 35–47. Universität Münster, July 2004. p. 105.

[7] M. Alt and S. Gorlatch. Future-based RMI: Optimizing compositions of remote method calls on the grid. In *Proceedings of Euro-Par 2003 Parallel Processing*, number 2790 in Lecture Notes in Computer Science, pages 427–430. Springer-Verlag, 2003. p. 84.

[8] G. Amdahl. Validity of the single processor approach to achieving large-scale computing capabilities. In *AFIPS Conference Proceedings*, volume 30, pages 483–485. Thomson Book Company, 1967. p. 125.

[9] IEEE VHDL Analysis and Standardization Group. IEEE standard VHDL language reference manual. Technical report, IEEE Computer Society, 2002. IEEE Std 1076-2002. p. 27.

[10] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994. p. 56.

[11] M.A. Arbib. Perceptual structures and distributed motor control. In *Handbook of Physiology: The Nervous System, II. Motor Control*, pages 1448–1480. MIT Press, 1981. p. 8.

[12] H. Baker and C. Hewitt. The incremental garbage collection of processes. In *Proceedings of the ACM Conference on AI and Programming Languages*, pages 55–59. Association for Computing Machinery, 1977. p. 84.

[13] K.E. Batcher. Sorting networks and their applications. In *Proceedings of the AFIPS Spring Joint Computer Conference*, volume 32, pages 307–314, 1968. p. 20.

[14] A.J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, 15(5):757–763, October 1966. p. 14.

[15] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, February 1984. p. 81.

[16] T. Blank. The maspar MP-1 architecture. In *Compcon Spring '90 digest of papers*, pages 20–24. IEEE Computer Society, February 1990. p. 13.

[17] S. Bleuler, M. Laumanns, L. Thiele, and E. Zitzler. PISA — a platform and programming language independent interface for search algorithms. In Carlos M. Fonseca, Peter J. Fleming, Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele, editors, *Evolutionary Multi-Criterion Optimization (EMO 2003)*, Lecture Notes in Computer Science, pages 494 – 508. Springer-Verlag, 2003. p. 119.

[18] G. Borgefors. Distance transforms in digital images. *Computer Vision, Graphics and Image Processing*, 34(3):344–371, June 1986. p. 17.

[19] S. Borkar. Design challenges of technology scaling. *IEEE Micro*, 19(4):23–29, 1999. p. 109.

[20] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC-5)*, pages 243–252. IEEE Computer Society Press, 1996. p. 55.

[21] J.M. Boyle. *Software reusability: vol. 1, concepts and models*, chapter "Abstract programming and program transformation – an approach to reusing programs", pages 361–413. ACM Press, 1989. p. 62.

[22] J.M. Boyle, T.J. Harmer, and V.L. Winter. *Modern software tools for scientific computing*, chapter "The TAMPR program transformation system: simplifying the development of numerical software", pages 353–372. Birkhauser Boston Inc., 1997. p. 62.

[23] M. Bravenboer and E. Visser. Concrete syntax for objects. Domain-specific language embedding and assimilation without restrictions. In D.C. Schmidt, editor, *Proceedings of the 19th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04)*, pages 365–383. ACM Press, October 2004. p. 47.

[24] H. Broers, W.Caarls, P.P. Jonker, and R. Kleihorst. Architecture study for smart cameras. In *Proceedings of the EOS Conference on Industrial Imaging and Machine Vision*, pages 39–49. European Optical Society, June 13-15 2005.

[25] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. *ACM Transactions on Graphics*, 23(3):777–786, August 2004. Special Issue: Proceedings of the 2004 SIGGRAPH Conference. pp. 40 and 50.

[26] J. Caarls. *Pose estimation for mobile devices and augmented reality*. PhD thesis, Delft University of Technology, 2008. To appear. p. 120.

[27] W. Caarls and P.P. Jonker. Benchmarks for smartcam development. In *Proceedings of ACIVS 2003 (Advanced Concepts for Intelligent Vision Systems)*, pages 81–86. Ghent University, September 2-5 2003.

[28] W. Caarls, P.P. Jonker, and H. Corporaal. SmartCam: Devices for embedded intelligent cameras. In Mariël Schweizer, editor, *Proceedings of the 3rd PROGRESS workshop on Embedded Systems, Utrecht, The Netherlands*, pages 14–17. Technology Foundation STW, October 24 2002.

[29] W. Caarls, P.P. Jonker, and H. Corporaal. SmartCam design framework. In Mariël Schweizer, editor, *Proceedings of the 4th PROGRESS workshop on Embedded Systems, Nieuwegein, The Netherlands*. Technology Foundation STW, October 22 2003.

[30] W. Caarls, P.P. Jonker, and H. Corporaal. Data- and task parallel image processing on a mixed SIMD-ILP platform using skeletons and asynchronous RPC. In Mariël Schweizer, editor, *Proceedings of the 5th PROGRESS workshop on Embedded Systems, Nieuwegein, The Netherlands*, pages 27–34. Technology Foundation STW, October 20 2004.

[31] W. Caarls, P.P. Jonker, and H. Corporaal. Skeletons and asynchronous RPC for embedded data- and task parallel image processing. In Katsushi Ikeuchi, editor, *Proceedings of the 9th IAPR Conference on Machine Vision Applications*, pages 384–387. International Association for Pattern Recognition, May 16-18 2005.

[32] W. Caarls, P.P. Jonker, and H. Corporaal. Algorithmic skeletons for stream programming in embedded heterogeneous parallel image processing applications. In *Proceedings of the 20th IEEE International Parallel and Distributed Processing Symposium*. IEEE Computer Society, April 26-29 2006. p. 101.

[33] W. Caarls, P.P. Jonker, and H. Corporaal. Skeletons and asynchronous RPC for embedded data- and task parallel image processing. *IEICE Transactions on Information and Systems*, E89-D(7), July 2006. p. 76.

[34] J. Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 8(6):679–698, November 1986. p. 120.

[35] J. Cocke and V. Markstein. The evolution of risc technology at ibm. *IBM Journal of Research and Development*, 34(1):4–11, 1990. p. 130.

[36] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989. ISBN 0-273-08807-6. p. 35.

[37] Y. Collette and P. Siarry. *Multiobjective Optimization: Principles and Case Studies*. Decision Engineering. Springer-Verlag, 2003. ISBN 3-540-40182-2. p. 118.

[38] R. Collins, A. Lipton, and T. Kanade. A system for video surveillance and monitoring. In *Proceedings of the American Nuclear Society 8th Internal Topical Meeting on Robotics and Remote Systems*, April 1999. p. 129.

[39] C. Consel, J.L. Lawall, and A-F. Le Meur. A tour of Tempo: a program specializer for the c language. *Science of Computer Programming*, 52(1–3):341–370, August 2004. p. 56.

[40] NVIDIA Corporation. Nvidia GeForce 8800 GPU architecture overview. Technical Report TB-02787-001_v01, NVIDIA Corporation, November 2006. `http://www.nvidia.com/object/IO_37100.html`. p. 13.

[41] G. Cybenko. Load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 7:279–301, 1989. p. 21.

[42] K. Czarnecki. Overview of generative software development. In *Proceedings of the workshop on Unconventional Programming Paradigms*, volume 3566 of *Lecture Notes in Computer Science*, pages 313–328. Springer-Verlag, September 2004. p. 53.

[43] K. Czarnecki and U. Eisenecker. *Generative Programming: Methods, Tools, and Applications*. Addison-Wesley Professional, 2000. ISBN 978-0201309775. p. 53.

[44] K. Czarnecki, U.W. Eisenecker, R. Glueck, D. Vandevoorde, and T.L. Veldhuizen. Generative programming and active libraries. In *Selected Papers from the International Seminar on Generic Programming*, volume 1766 of *Lecture Notes in Computer Science*, pages 25–39. Springer-Verlag, 1998. p. 47.

[45] L. Dagum and R. Menon. OpenMP: An industry-standard API for shared-memory programming. *IEEE Computational Science and Engineering*, 5(1):46–55, January/March 1998. p. 22.

[46] M. Danelutto, R. Di Meglio, S. Orlando, S. Pelagatti, and M. Vanneschi. A methodology for the development and the support of massively parallel programs. *Future Generation Computer Systems*, 8(1–3), July 1992. p. 79.

[47] R. Eigenmann. Programming distributed memory sytems using openmp. In *Proceedings of the 12th International Workshop on High-Level Parallel Programming Models and Supportive Environments*. IEEE Computer Society, March 2007. Held in conjunction with IPDPS 2007. p. 23.

[48] D.R. Engler. Interface compilation: Steps toward compiling program interfaces as languages. *IEEE Transactions on Software Engineering*, 25(3):387–400, May 1999. p. 47.

[49] H. Fatemi. *Processor Architecture Design for Smart Cameras*. PhD thesis, Eindhoven University of Technology, 2007. ISBN 978-90-386-1983-5. p. 13.

[50] M.J. Flynn. Very high-speed computing systems. *Proceedings of the IEEE*, 54(12):1901–1909, December 1966. p. 9.

[51] M.J. Flynn and P. Hung. Microprocessor design issues: thoughts on the road ahead. *IEEE Micro*, 25(3):16–31, May-June 2005. p. 8.

[52] High Performance Fortran Forum. *High Performance Fortran Language Specification*, 1993. Version 1.0. p. 27.

[53] Message Passing Interface Forum. MPI: A message-passing interface standard. *International Journal of Supercomputing Applications and High Performance Computing*, 8(3/4):159–416, 1994. p. 23.

[54] T.J. Fountain. *Machine Vision: Algorithms, Architectures, and Systems*, chapter "Introducing Local Autonomy to Processor Arrays", pages 31–56. Academic Press, 1988. p. 9.

[55] Y. Fujita, S. Kyo, N. Yamashita, and S. Okazaki. A 10 GIPS SIMD processor for PC-based real-time vision applications – architecture, algorithm implementation and language support –. In C.C. Weems Jr., editor, *Proceedings of the 4th IEEE International Workshop on Computer Architecture for Machine Perception*, pages 22–32. IEEE Computer Society, 1997. p. 25.

[56] Y. Futamura. Partial evaluation of computation process - an approach to a compiler-compiler. *Systems, Computers, Controls*, 2(5):45–50, 1971. p. 65.

[57] H.L. Gantt. *Work, Wages, and Profits*. Hive Publishing Corporation, 1974. Originally published in the *New York Engineering Magazine* (1910). p. 82.

[58] D.E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., 1989. p. 118.

[59] R. Gonzalez, B.M. Gordon, and M.A. Horowitz. Supply and threshold voltage scaling for low power cmos. *IEEE Journal of Solid-State Circuits*, 32(8):1210–1216, August 1997. p. 109.

[60] A. Grama, A. Gupta, G. Karypis, and V. Kumar. *Introduction to Parallel Computing (second edition)*. Addison-Wesley, 2003. p. 20.

[61] M. Gries. Methods for evaluating and covering the design space during early design development. *The VLSI Journal of Integration*, 38(2):131–183, December 2004. p. 107.

[62] T. Grötker, S. Liao, G. Martin, and S. Swan. *System Design with SystemC*. Springer-Verlag, 2002. ISBN 978-1-4020-7072-3. p. 125.

[63] IEEE Verilog Standards Group. IEEE standard for Verilog hardware description language. Technical report, IEEE Computer Society, 2006. IEEE Std 1364-2005. p. 27.

[64] G.M. Hagen, W. Caarls, M. Thomas, A. Hill, K.A. Lidke, B. Rieger, C. Fritsch, B. van Geest, T.M. Jovin, and D.J. Arndt-Jovin. Biological applications of an LCoS-based programmable array microscope (PAM). In *Proceedings of SPIE*, volume 6441. SPIE, January 2007.

[65] B. Haible. ffcall - foreign function call libraries, version 1.10. `http://www.haible.de/bruno/packages-ffcall.html`. p. 70.

[66] R.H. Halstead, jr. Multilisp: A language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, October 1985. p. 84.

[67] J. Henkel. Closing the SoC design gap. *IEEE Computer*, 36:119–221, 2003. p. 7.

[68] W.D. Hillis. *The Connection Machine*. PhD thesis, Massachusetts Institute of Technology, 1985. ISBN 0-262-08157-1. p. 13.

[69] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985. ISBN 0-13-153271-8. p. 27.

[70] C.A.R. Hoare. *Occam 2 Reference Manual*. INMOS Limited, 1988. p. 27.

[71] K.E. Iverson. *A programming language*. John Wiley and Sons, Ltd., 1962. p. 36.

[72] P. Klint J. Heering, P.R.H. Hendriks and J. Rekers. The syntax definition formalism sdf — reference manual. *ACM SIGPLAN Notices*, 24:43–75, 1989. p. 58.

[73] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation*. Prentice Hall International, June 1993. ISBN 0-13-020249-5. Available online at `http://www.dina.kvl.dk/~sestoft/pebook/pebook.html`. p. 56.

[74] S.P. Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2002. p. 35.

[75] P.P. Jonker and W. Caarls. Application driven design of embedded real-time image processors. In *Proceedings of ACIVS 2003 (Advanced Concepts for Intelligent Vision Systems)*, pages 1–8. Ghent University, September 2-5 2003.

[76] G. Kahn. The semantics of a simple language for parallel programming. In *Proceeedings of IFIP Congress 74*, pages 471–475. North-Holland, August 1974. p. 99.

[77] B.W. Kernighan and D.M. Ritchie. The M4 macro processor. Technical report, Bell Laboratories, Murray Hill, New Jersey, USA, 1977. p. 55.

[78] Brucek Khailany. *The VLSI Implementation and Evaluation of Area- and Energy-Efficient Streaming Media Processors*. PhD thesis, Stanford University, June 2003. p. 41.

[79] G. A. P. Kindervater and H. W. J. M. Trienekens. Experiments with parallel algorithms for combinatorial problems. *European Journal of Operational Research*, 33(1):65–81, 1988. p. 21.

[80] R. Kleihorst, H. Broers, A. Abbo, H. Embrahimmalek, H. Fatemi, H. Corporaal, and P. Jonker. An SIMD-VLIW smart camera architecture for real-time face recognition. In *Proceedings of ProRISC 2003*, pages 1–7. Technology Foundation STW, November 26-27 2003. p. 89.

[81] R. Kleihorst, B. Schueler, A. Danilin, and M. Heijligers. Smart camera mote with high performance vision system. In *Proceedings of the ACM SenSys 2006 Workshop on Distributed Smart Cameras*, October 2006. p. 129.

[82] D. Koelma and A. Smeulders. Visual programming interface for an image-processing environment. *Pattern Recognition Letters*, 15(11):1099–1109, November 1994. p. 128.

[83] E.R. Koomen. *Low-level Image Processing Architectures*. PhD thesis, Delft University of Technology, 1990. ISBN 90-9003713-6. p. 9.

[84] H. Kuchen. A skeleton library. In B. Monien and R. Feldman, editors, *Proceedings of the 8th International Euro-Par Conference*, volume 2400 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. p. 55.

[85] S. Kyo. A 51.2GOPS programmable video recognition processor for vision based intelligent cruise control applications. In K. Ikeuchi, editor, *Proceedings of the 2002 MVA Workshop*, pages 632–635. International Association for Pattern Recognition, December 11–13 2002. p. 12.

[86] S. Kyo and K. Sato. Efficient implementation of image processing algorithms in linear processor arrays using the data parallel language 1DC. In M. Takagi, editor, *Proceedings of the 1996 MVA Workshop*, pages 160–165. International Association for Pattern Recognition, 1996. p. 25.

[87] P. Lieverse, P. van der Wolf, K. Vissers, and E. Deprettere. A methodology for architecture exploration of heterogeneous signal processing systems. *Journal of VLSI signal processing*, 23(3):197–207, 2001. p. 125.

[88] B. Liskov and L. Shira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *Proceedings of the ACM SIGPLAN 1988 conference on Programming Language design and Implementation*, pages 260–267. ACM Press, 1988. p. 84.

[89] D.G. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. p. 101.

[90] H. Makholm. Specializing c - an introduction to the principles behind c-mix ii. Technical report, DIKU, University of Copenhagen, 1999. p. 56.

[91] F. Mantz, P.P. Jonker, and W. Caarls. Behavior-based vision on a 4 legged soccer robot. In A. Bredenfeld, A. Jacoff, I. Noda, and Y. Takahashi, editors, *Proc. 9th RoboCup International Symposium, Osaka, Japan*, volume 4020 of *Lecture Notes in Computer Science*, pages 480–487. Springer-Verlag, July 2006. p. 8.

[92] H. Masuhara, S. Matsuoka, K. Asai, and A. Yonezawa. Compiling away the meta-level in object-oriented concurrent reflective languages using partial evaluation. In *Proceedings of OOPSLA'95*, pages 300–315. ACM SIGPLAN, 1995. p. 66.

[93] The Mathworks, Inc., 3 Apple Hill Drive, Natpick, MA 01760-2098. *Video and Image Processing Blockset 2*, 2.3 (2007a) edition, March 2007. p. 128.

[94] P. Mattson. *A Programming System for the Imagine Media Processor*. PhD thesis, Dept. of Electrical Engineering, Stanford University, 2001. p. 41.

[95] J.L. McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Communications of the ACM*, 3(4):184–195, 1960. p. 36.

[96] M. McCool and S. Du Toit. *Metaprogramming GPUs with Sh*. A.K. Peters, Ltd., 2004. ISBN 1-56881-229-9. p. 55.

[97] G. Michaelson, N. Scaife, P. Bristow, and P. King. Nested algorithmic skeletons from higher order functions. *Parallel Algorithms and Applications special issue on High Level Models and Languages for Parallel Processing*, 16:181–206, August 2001. p. 79.

[98] S. Mouy. Xtc2 language syntax. Technical report, Philips Electronics, 2006. p. 27.

[99] M. Nijhuis, H. Bos, and H.E. Bal. Supporting reconfigurable parallel multimedia applications. In *Proc. Euro-PAR '06, Dresden, Germany*, pages 765–776, August 2006. pp. 42 and 46.

[100] J.G.E. Olk. *Distributed Bucket Processing - A paradigm for parallel image processing*. PhD thesis, Delft University of Technology, ASCI, September 2001. ASCI Dissertation Series 68. p. 18.

[101] T.M. Parks. *Bounded Scheduling of Process Networks*. PhD thesis, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley, December 1995. Technical Report UCB/ERL-95-105. p. 100.

[102] T.J. Parr and R.W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software Practice and Experience*, 25(7):789–810, July 1995. p. 56.

[103] P.Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. In *Proc. ninth annual conference on Object-oriented programming systems, language, and applications*, pages 341–354. ACM Press, 1994. p. 84.

[104] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Transactions on Computers*, 55(2):99–112, February 2006. p. 125.

[105] M. Poletto, W.C. Hsieh, D. Engler, and M.F. Kaashoek. 'c and tcc: A language and compiler for dynamic code generation. *ACM Transactions on Programming Languages and Systems*, 21(2), March 1999. pp. 57 and 59.

[106] W.V.O. Quine. *Mathematical Logic*. Harvard University Press, 1940. p. 55.

[107] F.A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer-Verlag, 2003. ISBN 1-85233-506-8. p. 35.

[108] A. Radulescu. *Compile-Time Scheduling for Distributed-Memory Suystems*. PhD thesis, Delft University of Technology, 2001. pp. 103 and 105.

[109] A.M. Rogers, J.H. Reppy, and L.J. Hendren. Supporting SPMD execution for dynamic data structures. In *Proceedings of the 5th International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 192–207. Springer-Verlag, August 1992. p. 84.

[110] C. Roig, A.Ripoll, M.A. Senar, F. Guirado, and E. Luque. A new model for static mapping of parallel applications with task and data parallelism. In *Proceedings of the 16th International Symposium on Parallel and Distributed Processing*, page 78. IEEE Computer Society, 2002. p. 105.

[111] M. Roth. Javaserver pages 2.0 specification. Technical Report SR-000152, Sun Microsystems, Inc., November 2003. p. 55.

[112] T. Sakurai and A.R. Newton. Alpha-power law mosfet model and its applications to cmos inverter delay and other formulas. *IEEE Journal of Solid-State Circuits*, 25(2):584–594, April 1990. p. 109.

[113] I. D. Scherson and S. Sen. Parallel sorting in two-dimensional vlsi models of computation. *IEEE Transactions on Computers*, 38(2):238–249, 1989. p. 20.

[114] F.J. Seinstra. *User Transparent Parallel Image Processing*. PhD thesis, University of Amsterdam, 2003. ISBN 90-5776-102-5. p. 46.

[115] M.A. Senar, A. Ripoll, A. Cortes, and E. Luque. Clustering and reassignment-based mapping strategy for message-passing architectures. *Journal of Systems Architecture: the EUROMICRO Journal*, 48(8-10):267–283, March 2003. p. 103.

[116] J. Serot, D. Ginhac, and J.-P. Derutin. SKiPPER: A skeleton-based parallel programming environment for real-time image processing applications. In V. Malyshkin, editor, *5th International Conference on Parallel Computing Technologies*, volume 1662 of *Lecture Notes in Computer Science*, pages 296–305. Springer-Verlag, September 1999. p. 79.

[117] T. Sheard. Accomplishments and research challenges in meta-programming. In *Proceedings of the Second International Workshop on Semantics, Applications, and Implementation of Program Generation*, pages 2–44. Springer-Verlag, 2001. pp. 53 and 54.

[118] G.A. Slavenburg. *TM1000 Databook*. TriMedia Division, Philips Semiconductors, 1997. p. 12.

[119] C. Soviany. *Embedding Data and Task Parallelism in Image Processing Applications*. PhD thesis, Delft University of Technology, 2003. ISBN 90-9016878-8. pp. 46 and 105.

[120] R. Stephens. A survey of stream processing. *Acta Informatica*, 34(7):491–541, 1997. p. 38.

[121] T. Sterling, D. Savarese, D. J. Becker, J. E. Dorband, U. A. Ranawake, and C. V. Packer. BEOWULF: A parallel workstation for scientific computation. In *Proceedings of the 24th International Conference on Parallel Processing*, pages I:11–14. CRC Press, 1995. p. 13.

[122] W. Taha and T. Sheard. Metaml and multi-stage programming with explicit annotations. *Theoretical Computer Science*, 248(1–2):211–242, October 2000. p. 57.

[123] W. Thies, M. Karczmarek, and S. Amarasinghe. Streamit: A language for streaming applications. In *International Conference on Compiler Construction*, volume 2304 of *Lecture Notes in Computer Science*. Springer-Verlag, 2002. p. 42.

[124] H. Topcuoglu, S. Hariri, and Min-You Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *IEEE Transactions on Parallel and Distributed Systems*, 13(3):260–275, September 2004. p. 103.

[125] W.J. Trybula. A common base for mask cost of ownership. In *Proceedings of the 23rd Annual BACUS Symposium on Photomask Technology*, volume 5256 of *Proceedings of SPIE*, pages 318–323. SPIE, 2003. p. 7.

[126] M.G.J. van den Brand, H.A. de Jong, P. Klint, and P.A. Olivier. Efficient annotated terms. *Software Practice and Experience*, 30:259–291, 2000. p. 65.

[127] J. van der Horst, R. van Leeuwen, H. Broers, R. Kleihorst, and P.P. Jonker. A real-time stereo SmartCam, using FPGA, SIMD and VLIW. In *Proceedings of the Second Workshop on Applications of Computer Vision*, pages 1–8, May 2006. Held in conjunction with ECCV 2006. p. 101.

[128] P. van der Wolf, E. de Kock, T. Henriksson, W. Kruijtzer, and G. Essink. Design and programming of embedded multiprocessors: An interface-centric approach. In *Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. IEEE/ACM/IFIP, September 2004. p. 46.

[129] A.J.C. van Gemund. *Performance Modeling of Parallel Systems*. PhD thesis, Delft University of Technology, 1996. p. 125.

[130] T. Veldhuizen. Expression templates. *C++ Report*, 5(7):26–31, June 1995. p. 55.

[131] T.L. Veldhuizen. Using C++ template metaprograms. *C++ Report*, 7(4):36–43, May 1995. pp. 47 and 55.

[132] E. Visser. *Syntax Definition for Language Prototyping*. PhD thesis, University of Amsterdam, September 1997. p. 58.

[133] E. Visser. Stratego: A language for program transformation based on rewriting strategies. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001. p. 61.

[134] E. Visser. Meta-programming with concrete object syntax. In Don Batory, Charles Consel, and Walid Taha, editors, *Generative Programming and Component Engineering (GPCE'02)*, volume 2487 of *Lecture Notes in Computer Science*, pages 299–315. Springer-Verlag, October 2002. p. 55.

[135] O. Wechsler. Inside Intel Core microarchitecture. Technical report, Intel Corporation, 2006. p. 12.

[136] A.K. Weissinger. *ASP in a Nutshell*. O'Reilly Media, Inc., July 2000. p. 55.

[137] N. Weste and D. Harris. *CMOS VLSI Design: A Circuits and Systems Perspective*. Addison-Wesley, 2004. p. 110.

[138] J.E. White. Rfc 707: A high-level framework for network-based resource sharing, 1976. p. 81.

[139] ISO/IEC JTC1/SC22/WG14 working group. Programming languages - C. Technical Report 9899:1999, International Standards Committee, December 1999. p. 59.

[140] Cheng-Zhong Xu and Francis C. M. Lau. Optimal parameters for load balancing with the diffusion method in mesh networks. *Parallel Processing Letters*, 4(1-2):139–147, 1994. p. 21.

[141] E. Zitzler, M. Laumanns, and L. Thiele. SPEA2: Improving the Strength Pareto Evolutionary Algorithm. Technical Report 103, Computer Engineering and Networks Laboratory (TIK), Swiss Federal Institute of Technology (ETH) Zurich, 2001. p. 118.

Appendix A

Modeling language

Our application and architecture models are textually represented in the same XML-based modeling language, called EPML (Environment-Property Modeling Language). It was designed to populate the central SmartCam data structure, and to be easily written both by hand and by the various scripts used in the SmartCam framework.

A.1 Data structure

The goal of the data structure, is to describe the relations and properties between remote functions, their arguments, the target architectures they run on, and the resources they need. This essentially creates a conventional relational database model, with extra tables for the properties of the relations themselves.

For example, an operation may have properties of itself, such as its name and function identifier. In the same vein, a target architecture has certain properties, like network address and idle power consumption. An operation may run on many targets, while a target may support many operations. Furthermore, the combination of the two, called a "target-operation", has properties of its own: the processing time and energy required for the operation to run on that particular architecture.

Figure A.1 illustrates the relations between the various objects. There are four basic types: Resources, Links, Targets and Operations. There are three combinations of two classes: Target Resources, Target Operations, and Operation Arguments (Argument is not a base type, as it has no properties of its own). Finally, there are two combinations of *three* classes: Target Operation Resources, and Target Operation Arguments.

A.2 Semantics

The language is built around the same concept as the data structure: objects with properties and object combinations with properties. We call a combination of one

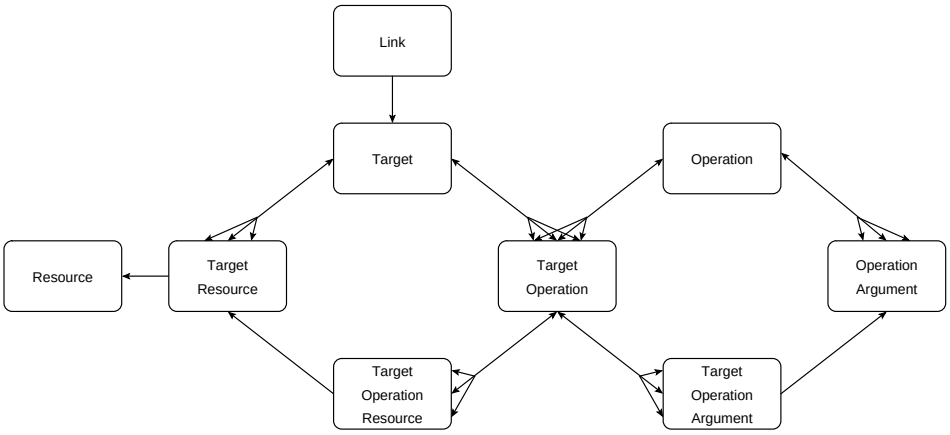


Figure A.1: Interactions between the various object classes in the central SMART-CAM data structure.

or more object types an *environment*, and within an environment, properties of that environment can be set.

Perhaps the easiest to understand way of specifying an environment is using the `env` tag, such as in program A.1. This code specifies that there is a target

Program A.1 Defining a `capture` operation on the `sensor` target using the `env` tag.

```
<env target="sensor" operation="capture">
  <E time="33m"/>
</env>
```

architecture called `sensor`, which can run an operation called `capture` in 33 ms (SI prefixes can be used when specifying numbers). It is not necessary to define the objects beforehand: they are created automatically. To specify that `capture` has an output argument that has a regular distribution on `sensor`, we may write as in program A.2. This way of specifying environments is easy for scripts,

Program A.2 Adding information about an operation argument on a specific target.

```
<env target="sensor" operation="capture" argument="o">
  <E timedist="regular"/>
</env>
```

but cumbersome for humans. We therefore have a more intuitive way, by *nesting* environments, illustrated in program A.3. Note that the order in which the environments are specified is irrelevant; it is only important *which* environments are there. This makes it easy to either make a list of targets and identify which operations can run on them, or to make a list of operations and define which targets support them. To further support this, we have special tags for creating lists of environments, such as in program A.4.

Program A.3 Nesting environments leads to a more easily readable specification.

```
<env target="sensor">
  <env operation="capture">
    <E time="33m"/>

    <env argument="o">
      <E timedist="regular"/>
    </env>
  </env>
</env>
```

Program A.4 Nested environment lists allow an even more intuitive approach to defining the model, by using objects names as tags.

```
<targets>
  <sensor>
    <!-- Sensor properties -->
  </sensor>
  <ilp>
    <operations>
      <gauss>
        <E time="0.1"/>
        <arguments>
          <i>
            <E timedist="regular"/>
          </i>
          <o>
            <E timedist="regular"/>
          </o>
        </arguments>
      </gauss>
      <display>
        <E time="16.7m"/>
      </display>
    </operations>
  </ilp>
</targets>
```

For an even shorter form, program A.5 shows that the properties of an environment in an environment list can also be specified immediately, inside the object's tag itself.

Program A.5 Specifying environment properties inside an object tag.

```
<targets>
  <ilp>
    <operations>
      <gauss time="0.1">
        <arguments>
          <i timedist="regular"/>
          <o timedist="regular"/>
        </arguments>
      </gauss>
      <display time="16.7m"/>
    </operations>
  </ilp>
</targets>
```

A.2.1 Environment hierarchies

A base object environment may be derived from another base object environment of the same type, provided that it is defined after its parent. Deriving means that all properties of the parent environment, as well as environments involving the parent as one of the constituents, apply to the derived environment, unless overwritten. An example is creating different processors of the same type, such as in program A.6.

A.2.2 Links

There is one special environment: the `link`, which connects `target`s together, signifying that they can communicate. Although this environment does not combine with the `target` environment to create a `link target`, it can be declared inside the target environment. This means that the target will act as a "default" link source or destination, and makes it easier to create networks. The two specifications in program A.7 are therefore identical.

A.3 Syntax

We have created an SDF specification of the modeling language, consisting of a lexical part (program A.8) and a context-free part (program A.9).

Program A.6 Environment hierarchies avoid duplicate work by allowing objects to derive from other objects.

```
<targets>
  <ilp>
    <operations>
      <gauss>
        <!-- Properties of gauss operation on ILP processors -->
      </gauss>
    </operations>
  </ilp>

  <ilp1>
    <E derives="ilp"/>
    <E address="/proc/ilp1"/>
  </ilp1>
  <master-ilp>
    <E derives="ilp"/>
    <E address="/proc/master"/>

    <!-- Extra operations available only to the master -->
    <operations>
      <display>
        <!-- Properties of display operation on master -->
      </display>
    </operations>
  </master-ilp>
</targets>
```

Program A.7 Defining a link environment inside a target, specifying a default source or destination. The equivalent code, but without nesting a link within a target, is to the right.

```
<targets>
  <sensor></sensor>
  <simd>
    <env link="sensor2simd">
      <E source="sensor"/>
    </env>
  </simd>
  <ilp>
    <env link="simd2ilp">
      <E source="simd"/>
    </env>
  </ilp>
</targets>
```

```
<targets>
  <sensor></sensor>
  <simd></simd>
  <ilp></ilp>
</targets>
<links>
  <sensor2simd>
    <E source="sensor"/>
    <E destination="simd"/>
  </sensor2simd>
  <simd2ilp>
    <E source="simd"/>
    <E destination="ilp"/>
  </simd2ilp>
</links>
```

Program A.8 Lexical syntax for EPML.

```
module epml-lexical
exports
  sorts
    Comment Dash DoubleDash
    IDENTIFIER VALUE TARGET OPERATION RESOURCE ARGUMENT LINK
    TYPE CLASS

  lexical syntax
    [\ \t\n\13]                                  -> LAYOUT

  context-free restrictions
    LAYOUT?                                      -/- [\ \t\n\13]

  lexical syntax
    "<!--" (~[\-] | Dash | DoubleDash)* "-->" -> Comment
    Comment                                      -> LAYOUT
    "-"                                          -> Dash
    "--"                                         -> DoubleDash

  lexical restrictions
    Dash                                         -/- [\-]
    DoubleDash                                   -/- [\>]

  lexical syntax
    [A-Za-z\_] [A-Za-z0-9\.\-\_]*                -> IDENTIFIER
    ~[\"\n]*                                     -> VALUE
    "target"                                     -> TARGET
    "operation"                                  -> OPERATION
    "resource"                                   -> RESOURCE
    "argument"                                   -> ARGUMENT
    "link"                                       -> LINK
    (TARGET|OPERATION|RESOURCE|ARGUMENT|LINK) -> TYPE
    TYPE "s"                                     -> CLASS

  lexical restrictions
    IDENTIFIER                                   -/- [A-Za-z0-9\.\-\_]
    VALUE                                        -/- ~[\"]
```

Program A.9 Context-free productions for EPML.

```
module epml
imports
  epml-lexical
exports
  context-free start-symbols
    Document
  sorts
    Id Value Type Class
    PropSpec ObjectSpec PropElement Env TypeList ListEnv Document

  context-free syntax
    IDENTIFIER                -> Id    {cons("Id")}
    VALUE                     -> Value {cons("Value")}
    TYPE                      -> Type  {cons("Type")}
    CLASS                     -> Class {cons("Class")}

  context-free syntax
    Id "=" "\"" Value "\""    -> PropSpec {cons("PropSpec")}
    Type "=" "\"" Id "\""     -> ObjectSpec {cons("ObjectSpec")}
    "<" "E" PropSpec* "/>"    -> PropElement {cons("PropElement")}

    "<" "env" ObjectSpec* ">"
      (TypeList|Env|PropElement)*
    "</" "env" ">"            -> Env {cons("Env")}
    "<" "env" ObjectSpec* "/>" -> Env {cons("EmptyEnv")}

    "<" Class ">"
      ListEnv*
    "</" Class ">"            -> TypeList {cons("TypeList")}
    "<" Id PropSpec* ">"
      (TypeList|Env|PropElement)*
    "</" Id ">"               -> ListEnv {cons("ListEnv")}
    "<" Id PropSpec* "/>"     -> ListEnv {cons("EmptyListEnv")}

    "<" "document" ">"
      (Document|TypeList|Env)*
    "</" "document" ">"       -> Document {cons("Document")}
```

Appendix B

Language syntax

We present the syntax of our two new languages as an SDF specification. SmartCam-C is the language used to specify the kernels, while PEPCIis the pseudo-dynamic meta-programming language used to instantiate them.

B.1 SmartCam-C

Program B.1 contains the additional productions understood by the SmartCam compiler, for the specification of kernels. They are: new storage classes for use in kernel arguments, kernel definition syntax, and a method for array *range* specification, for stream stencilling. There is also a rule to allow `code` arguments to functions to be specified without explicit quotation, allowing constructs such as "`init { /* code */ };`".

B.2 PEPCI

PEPCI adds two keywords, a quotation and anti-quotation mechanism, code evaluation, and embedded Stratego. The productions for these are shown in program B.2.

B.2.1 Additional syntax for the 1DC language

Extending the PEPCI syntax with additional target languages is quite easy because of the modularity of SDF. Program B.3 lists the additional productions needed to support the 1DC language used for programming the NEC IMAP line of SIMD processors.

Program B.1 Productions for kernel definition in SMARTCAM-C.

```
module scc
imports
  C
exports
  sorts StorageClass
  context-free syntax
    "in"             -> StorageClass {cons("In")}
    "out"            -> StorageClass {cons("Out")}
    "inout"          -> StorageClass {cons("InOut")}
    "stream"         -> StorageClass {cons("Stream")}
    "lookup"         -> StorageClass {cons("Lookup")}

  sorts ExternalDef PostfixExpArrayRange
  context-free syntax
    FunDefDecl FunDefDecl CompoundStm
      -> ExternalDef {cons("KernelDef")}
    PostfixExp "[" CommaExp ".." CommaExp "]"
      -> PostfixExp {cons("ArrayRange")}
    FunCall CompoundStm+ -> FunCallCode {cons("FunCallCode")}
```

Program B.2 PEPCI language definition.

```
module PEPCI
  imports C StrategoRenamed
exports
  sorts Keyword
  lexical syntax
    "code"                -> Keyword
    "promote"             -> Keyword

  sorts BasicTypeName StorageClass
  context-free syntax
    "code"                -> BasicTypeName {cons("Code")}
    "promote"             -> StorageClass {cons("Promote")}

  sorts Code
  context-free syntax
    TypeName              -> AllowableCode {avoid}
    CommaExp              -> AllowableCode
    Stm                   -> AllowableCode
    Declaration           -> AllowableCode
    "`" AllowableCode "`" -> Code {cons("CodeConst")}

  sorts PrimaryExp UnaryExp TypedefName
  context-free syntax
    Code -> PrimaryExp
    "@" CastExp -> IDENTIFIER {cons("Decode"), avoid}
    "@" CastExp -> UnaryExp {cons("Decode"), prefer}
    "@" CastExp -> TypedefName {cons("TypeDecode"), avoid}
    "$" CastExp -> UnaryExp {cons("Escape")}

  sorts StrategoDef
  context-free syntax
    FunDefTypeSpec FunDefDecl "{" StrategoDecl* "}"
      -> FunDef {cons("StrategoDef"), avoid}
```

Program B.3 Productions for supporting the 1DC language.

```
module NEC
imports PEPCI
exports
  sorts Keyword
  lexical syntax
    "sep"      -> Keyword
    "separate" -> Keyword
    "outside"  -> Keyword
    "mif"      -> Keyword
    "melse"    -> Keyword
    "mdo"      -> Keyword
    "mwhile"   -> Keyword
    "mfor"     -> Keyword

  sorts
    StorageClass PostfixExp FunCall UnaryExp ShiftExp
    SelectionStm IterationStm
  context-free syntax
    "sep"      -> StorageClass {cons("Separate")}
    "separate" -> StorageClass {cons("Separate")}
    "outside"  -> StorageClass {cons("Outside")}

    PostfixExp ":[" CommaExp ":]"
      -> PostfixExp {cons("NECArrayIndex")}

    ":<" CastExp  -> UnaryExp {cons("NECUnaryShiftLeft")}
    ":>" CastExp  -> UnaryExp {cons("NECUnaryShiftRight")}
    ":&&" CastExp -> UnaryExp {cons("NECGlobalAnd")}
    ":||" CastExp -> UnaryExp {cons("NECGlobalOr")}
    ShiftExp ":<" AddExp -> ShiftExp {cons("NECBinaryShiftLeft")}
    ShiftExp ":>" AddExp -> ShiftExp {cons("NECBinaryShiftRight")}

    "mif" "(" CommaExp ")" Stm
      -> SelectionStm {cons("NECIf")}
    "mif" "(" CommaExp ")" Stm "melse" Stm
      -> SelectionStm {cons("NECIfElse")}
    "mwhile" "(" CommaExp ")" Stm
      -> IterationStm {cons("NECWhile")}
    "mdo" Stm "mwhile" "(" CommaExp ")" ";"
      -> IterationStm {cons("NECDoWhile")}
    "mfor" "(" CommaExpOpt ";" CommaExpOpt ";" CommaExpOpt ")" Stm
      -> IterationStm {cons("NECFor")}
```

Appendix C

Examples

We give one example each of a SmartCam-C stream program (including both kernel definition and stream coordination), a skeleton, and a meta-skeleton. Please recall that only the stream program is written by regular users. Skeletons and especially meta-skeletons are the realm of specialized programmers.

C.1 Stream program

```
/*
>> Edge detection for AR pose estimation
>> Based on Caarls, J., "Pose estimation for mobile devices and
>> augmented reality" (to appear in 2008)
*/

#include "sc.h"
#include <math.h>

/* Gradient in the X direction */
NeighbourhoodToPixelOp()
gradx(in  stream unsigned char i[-1..1][-3..3],
      out stream short int *o)
{
  int val = 0;
  signed char kernel[3][7];
  int y, x;

  kernel = {{   3,  26,  59,  0, -59, -26,  -3},
            {   5,  43,  97,  0, -97, -43,  -5},
                3,  26,  59,  0, -59, -26,  -3}};

  for (y=-1; y <= 1; y++)
    for (x=-3; x <= 3; x++)
      val += i[y][x] * (int)kernel[y+1][x+3];

  *o = val/1024;
```

```
}

/* Gradient in the Y direction */
NeighbourhoodToPixelOp()
grady(in  stream unsigned char i[-3..3][-1..1],
      out stream short int *o)
{
  int val = 0;
  signed char kernel[7][3];
  int y, x;

  kernel = {{  3,   5,   3}, { 26,  43,  26}, { 59,  97,  59},
            {  0,   0,   0},
            {-59, -97, -59}, {-26, -43, -26}, { -3,  -5,  -3}};

  for (y=-3; y <= 3; y++)
    for (x=-1; x <= 1; x++)
      val += i[y][x] * (int)kernel[y+3][x+1];

  *o = val/1024;
}

/* Gradient magnitude (squared) */
PixelToPixelOp()
gradmag(in  stream short int *gx,
        in  stream short int *gy,
        out stream unsigned char *o)
{
  short int ax = *gx, ay = *gy, m;

  if (ax < 0) ax = -ax;
  if (ay < 0) ay = -ay;

  m = ax*ax+ay*ay;
  if (m > 255) m = 255;

  *o = m;
}

/* Determine a threshold using a histogram */
GlobalOp()
histthresh(in  stream unsigned char **g,
           in  double *p,
           out int *t)
{
  int hist[256], cum=0;
  int x, y, i;
  int target = *p*_stride*_lines;

  memset(hist, 0, 256*sizeof(int));
```

```
  for (y=0; y < _lines; y++)
    for (x=0; x < _stride; x++)
      hist[g[y][x]]++;

  for (i=255; i > 0 && cum < target; i--)
    cum += hist[i];

  *t = i;
}

/* Binarization */
PixelToPixelOp()
binarize(in  stream unsigned char *i,
         in  int *t,
         out stream unsigned char *o)
{
  *o = (*i > *t) * 255;
}

/* Edge orientation */
PixelToPixelOp()
orient(in  stream unsigned char *i,
       in  stream short int *gx,
       in  stream short int *gy,
       out stream unsigned char *o)
{
  if (*i)
  {
    double q = atan2(*gy, *gx);
    if (q < 0) q = M_PI + q;
    *o = q/M_PI*255;
  }
  else
    *o = 0;
}

/* Non-maximum suppression */
NeighbourhoodToPixelOp()
nonmax(in  stream unsigned char i[0][0],
       in  stream unsigned char g[-1..1][-1..1],
       in  stream unsigned char d[0][0],
       out stream unsigned char *o)
{
  if (i[0][0])
  {
    float lg = g[0][0], ld = d[0][0]/255.0*M_PI;

    *o = 255;

    if (ld < atan2(0.5, 1.5))
    {
```

```
      if (g[ 0][-1] > lg || g[ 0][ 1] > lg)
        *o = 0;
    }
    else if (ld < atan2(1.5, 0.5))
    {
      if (g[-1][-1] > lg || g[ 1][ 1] > lg)
        *o = 0;
    }
    else if (ld < atan2(1.5, -0.5))
    {
      if (g[-1][ 0] > lg || g[ 1][ 0] > lg)
        *o = 0;
    }
    else if (ld < atan2(0.5, -1.5))
    {
      if (g[ 1][-1] > lg || g[-1][ 1] > lg)
        *o = 0;
    }
    else
    {
      if (g[ 0][-1] > lg || g[ 0][ 1] > lg)
        *o = 0;
    }
  }
  else
    *o = 0;
}

/* Cornerness detection */
NeighbourhoodToPixelOp()
detcorner(in stream unsigned char b[0][0],
          in stream short int gx[-1..1][-1..1],
          in stream short int gy[-1..1][-1..1],
          out stream unsigned char *o)
{
  if (b[0][0])
  {
    int y, x;
    float dxdx=0, dydy=0, dxdy=0, tr, det, res;

    for (y=-1; y <= 1; y++)
      for (x=-1; x <= 1; x++)
      {
        dxdx += gx[y][x]*gx[y][x];
        dydy += gy[y][x]*gy[y][x];
        dxdy += gx[y][x]*gy[y][x];
      }

    tr = dxdx + dydy;
    det = dxdx*dydy-dxdy*dxdy;
```

```
    res = (4*det/(tr*tr))*255;
    if (res < 0) res = -res;
    if (res > 255) res = 255;

    *o = res;
  }
  else
    *o = 0;
}

/* Thinning */
RecursiveNeighbourhoodToPixelOp()
thin(in  stream unsigned char i[0..1][-1..1],
     out stream unsigned char o[-1..0][-1..1])
{
  unsigned char thin[256], nbh;

  init {
    int i, b;

    for (i = 0; i < 256; i++)
    {
      unsigned char mask[8];
      char hist[8];
      int num = 0, changed=1;

      for (b = 0; b < 8; b++)
      {
        if (i & (1 << b))
          mask[b] = b;
        else
          mask[b] = 255;
        hist[b] = 0;
      }

      while (changed)
      {
        changed = 0;
        for (b = 0; b < 8; b += 2)
        {
          // corners
          if (mask[b] != 255)
          {
            if (mask[b] > mask[(b + 1) % 8])
            {
              mask[b] = mask[(b + 1) % 8];
              changed = 1;
            }
            if (mask[b] > mask[(b - 1 + 8) % 8])
            {
              mask[b] = mask[(b - 1 + 8) % 8];
```

```
            changed = 1;
          }
        }
      }
      for (b = 1; b < 8; b += 2)
      {
        // 4-connected
        if (mask[b] != 255)
        {
          if (mask[b] > mask[(b + 1) % 8])
          {
            mask[b] = mask[(b + 1) % 8];
            changed = 1;
          }
          if (mask[b] > mask[(b - 1 + 8) % 8])
          {
            mask[b] = mask[(b - 1 + 8) % 8];
            changed = 1;
          }

          if (mask[b] > mask[(b + 2) % 8])
          {
            mask[b] = mask[(b + 2) % 8];
            changed = 1;
          }
          if (mask[b] > mask[(b - 2 + 8) % 8])
          {
            mask[b] = mask[(b - 2 + 8) % 8];
            changed = 1;
          }
        }
      }
    }
    for (b = 0; b < 8; b++)
      if (mask[b] != 255)
        hist[mask[b]]++;
    for (b = 0; b < 8; b++)
      if (hist[b])
        num++;
    if (num > 1)
      thin[i] = 255;
    else
      thin[i] = 0;
  }
};

if (i[0][0])
{
  nbh = ((o[-1][-1]>0)<<0)|((o[-1][ 0]>0)<<1)|((o[-1][ 1]>0)<<2)|
        ((o[ 0][-1]>0)<<7)|                   ((i[ 0][ 1]>0)<<3)|
        ((i[ 1][-1]>0)<<6)|((i[ 1][ 0]>0)<<5)|((i[ 1][ 1]>0)<<4);
```

```
    if (thin[nbh])
      o[0][0] = 255;
    else
      o[0][0] = 0;
  }
  else
    o[0][0] = 0;
}

MAINTYPE MAIN(int argc, char **argv)
{
  Future<STREAM> img, gx, gy, gm, bin, o, c;
  Future<int>    t;

  scInit(argc, argv);

  while (1)
  {
    /*          IN    IN           IN   OUT */
    capture   (                          &img);
    gradx     (&img,                     &gx );
    grady     (&img,                     &gy );
    gradmag   (&gx,  &gy,                &gm );
    histthresh(&gm,  scdbl(0.1),         &t  );
    binarize  (&gm,  &t,                 &bin);
    orient    (&bin, &gx,         &gy,   &o  );
    nonmax    (&bin, &gm,         &o,    &bin);
    thin      (&bin,                     &bin);
    detcorner (&bin, &gx,         &gy,   &c  );

    /* Continue with edge following on bin and c */
  }

  return scExit();
}
```

C.2 Skeleton

```
/*
>> Pixel to pixel operation for sequential targets
*/

#include "stdstratego.h"
#include "stdsc.h"

#include "pixelbypixel.h"

void PixelToPixelOp(argcode_t *argcode, int arguments, code body)
{
```

```
  int ii;

  for (ii=0; ii < arguments; ii++)
  {
    if (argcode[ii].argtype == ARG_STREAM_IN ||
        argcode[ii].argtype == ARG_STREAM_OUT)
    {
      @declare(argcode[ii].datatype, argcode[ii].id);

      body = replace(`*$@argcode[ii].id`, argcode[ii].id, body);
    }
    else
    {
      @declare(addrpointer(argcode[ii].datatype), argcode[ii].id);

      if (@dataclass(argcode[ii].datatype) == DATACLASS_INTEGER)
        @argcode[ii].id = &(@argcode[ii].argid).data.d;
      else if (@dataclass(argcode[ii].datatype) == DATACLASS_FLOAT)
        @argcode[ii].id = &(@argcode[ii].argid).data.f;
      else
        @argcode[ii].id = @cast(addpointer(argcode[ii].datatype),
                                (@argcode[ii].argid).data.b);
    }

    argcode[ii].state.buffer = 1;
    argcode[ii].state.state = 0;
    argcode[ii].state.delay = 0;
  }

  repeat {
    $@body;
  };

  epilogue {
    int ii;

    for (ii=0; ii < arguments; ii++)
    {
      if (argcode[ii].argtype == ARG_STREAM_IN)
        BufferRemoveReference(ii);
      else if (argcode[ii].argtype == ARG_STREAM_OUT)
        BufferFinalize(ii);
    }
  };
}
```

C.3 Meta-skeleton

```
/*
>> pixelbypixel, metaskeleton for pixel operations
```

```
*/

#include "stdstratego.h"
#include "stdsc.h"

#include "ilpstrategies.h"
#include "linebyline.h"

#define PREPROLOGUE 0
#define PROLOGUE    1
#define REPEAT      2
#define EPILOGUE    3

typedef struct skelcode_s
{
  code body;
  code split[];
} skelcode_t;

code metasplit(code body)
{
  strategies
    main =
        <table-put>("globals", "prologue", Compound([], []))
      ; <table-put>("globals", "repeat", Compound([], []))
      ; <table-put>("globals", "epilogue", Compound([], []))
      ; <topdown(try(MetaSplit <+ AddPromote))>(body)
      ; ![<id>, <table-get>("globals", "prologue")
              , <table-get>("globals", "repeat")
              , <table-get>("globals", "epilogue")
         ]
      ; map(\a -> AssignInit(CodeConst(a))\); \a -> ArrayInit(a)\; id

  rules
    MetaSplit:
      Stat(FunCall(Id(i), [CodeConst(c)])) -> Stat(EmptyExp)
      where ( <elem>(i, ["prologue", "repeat", "epilogue"])
              ; <table-put>("globals", i, c)
            )

    AddPromote:
      Declaration2(t@TypeSpec(a, b, c),i) ->
      Declaration2(DeclSpec([Promote], t), i)

    AddPromote:
      Declaration2(DeclSpec(d, t), i) ->
      Declaration2(DeclSpec([Promote|d], t), i)
}

static int instanceBufferGetStride(int ii, int aa)
{
```

```
  if (instance[ii].argcode[aa].connector.instance == -1)
    return
      scuBufferGetStride(args[instance[ii].argcode[aa].
                             connector.argument].buffer);
  else
    error("Interactions with intermediate buffers not allowed");
}

static int instanceBufferGetLines(int ii, int aa)
{
  if (instance[ii].argcode[aa].connector.instance == -1)
    return
      scuBufferGetLines(args[instance[ii].argcode[aa].
                             connector.argument].buffer);
  else
    error("Interactions with intermediate buffers not allowed");
}

static int instanceBufferRemoveReference(int ii, int aa)
{
  if (instance[ii].argcode[aa].connector.instance == -1)
    return
      scuBufferRemoveReference(args[instance[ii].argcode[aa].
                                    connector.argument].buffer);
  else
    return 0;
}

static int instanceBufferFinalize(int ii, int aa)
{
  if (instance[ii].argcode[aa].connector.instance == -1)
    return
      scuBufferFinalize(args[instance[ii].argcode[aa].
                             connector.argument].buffer);
  else
    return 0;
}

static int instanceBufferPeek(int ii, int aa, void **data,
                              int size, int options)
{
  if (instance[ii].argcode[aa].connector.instance == -1)
    return
      scuBufferPeek(args[instance[ii].argcode[aa].
                         connector.argument].buffer,
                    args[instance[ii].argcode[aa].
                         connector.argument].reader,
                    data, size, options);
  else
    error("Interactions with intermediate buffers not allowed");
}
```

```
static void instanceBufferReleasePeeked(int ii, int aa, int size)
{
  if (instance[ii].argcode[aa].connector.instance == -1)
    scuBufferReleasePeeked(args[instance[ii].argcode[aa].
                             connector.argument].buffer,
                           args[instance[ii].argcode[aa].
                             connector.argument].reader,
                           size);
  else
    error("Interactions with intermediate buffers not allowed");
}

static int instanceBufferAllocate(int ii, int aa, void **data,
                                  int size, int options)
{
  if (instance[ii].argcode[aa].connector.instance == -1)
    return
      scuBufferAllocate(args[instance[ii].argcode[aa].
                         connector.argument].buffer,
                        data, size, options);
  else
    error("Interactions with intermediate buffers not allowed");
}

static void instanceBufferReleaseAllocated(int ii, int aa, int size)
{
  if (instance[ii].argcode[aa].connector.instance == -1)
    scuBufferReleaseAllocated(args[instance[ii].argcode[aa].
                                connector.argument].buffer,
                              size);
  else
    error("Interactions with intermediate buffers not allowed");
}

static code rewriteBuffers(int ii, code body)
{
  body = rewrite(‘BufferGetStride(PH1)‘,
                 ‘instanceBufferGetStride($ii, PH1)‘, body);
  body = rewrite(‘BufferGetLines(PH1)‘,
                 ‘instanceBufferGetLines($ii, PH1)‘,  body);
  body = rewrite(‘BufferRemoveReference(PH1)‘,
                 ‘instanceBufferRemoveReference($ii, PH1)‘, body);
  body = rewrite(‘BufferFinalize(PH1)‘,
                 ‘instanceBufferFinalize($ii, PH1)‘, body);
  body = rewrite(‘BufferPeek(PH1, PH2, PH3, PH4)‘,
                 ‘instanceBufferPeek($ii, PH1, PH2, PH3, PH4)‘,
                 body);
  body = rewrite(‘BufferReleasePeeked(PH1, PH2)‘,
                 ‘instanceBufferReleasePeeked($ii, PH1, PH2)‘,
                 body);
```

```
  body = rewrite(‘BufferAllocate(PH1, PH2, PH3, PH4)‘,
                 ‘instanceBufferAllocate($ii, PH1, PH2, PH3, PH4)‘,
                 body);
  body = rewrite(‘BufferReleaseAllocated(PH1, PH2)‘,
                 ‘instanceBufferReleaseAllocated($ii, PH1, PH2)‘,
                 body);

  return body;
}

void pixelbypixel(instance_t *instance, int instances)
{
  skelcode_t skelcode[];
  int ii, jj, kk, fii=-1, fia=-1;
  int pbp_hpixels, pbp_ii, stride;

  for (ii=0; ii < instances; ii++)
  {
    /* Make sure function calls in registered codes are expanded */
    instance[ii].body = rewrite(‘prologue(PH1)‘,
                                ‘prologue(reduce(PH1, 1))‘,
                                instance[ii].body);
    instance[ii].body = rewrite(‘repeat(PH1)‘,
                                ‘repeat(reduceit(PH1, 1))‘,
                                instance[ii].body);
    instance[ii].body = rewrite(‘epilogue(PH1)‘,
                                ‘epilogue(reduce(PH1, 1))‘,
                                instance[ii].body);

    /* Add skeleton functions to symbol table */
    @addpromote(instance[ii].body);

    /* Call skeleton, keeping expanded code */
    skelcode[ii].body =
      reduce(‘($@instance[ii].skeleton)(instance[ii].argcode,
                                        instance[ii].arguments,
                                        instance[ii].kernel)‘);

    /* Extract registered codes, and make variables unique */
    skelcode[ii].split = @metasplit(skelcode[ii].body);
    skelcode[ii].split = @scramblevariables(skelcode[ii].split,
                                            makeid(‘i‘, ii));

    /* Rewrite buffers to add instance information */
    for (jj=0; jj < 4; jj++)
      if (jj != REPEAT)
        skelcode[ii].split[jj] =
          rewriteBuffers(ii, skelcode[ii].split[jj]);

    /* Rename arguments to match scrambled variables */
    for (jj=0; jj < instance[ii].arguments; jj++)
```

```
  {
    instance[ii].argcode[jj].id =
      makeid(makeid(makeid(‘i‘, ii),
                    instance[ii].argcode[jj].id), ‘0‘);

    if (instance[ii].argcode[jj].connector.instance == -1 &&
        instance[ii].argcode[jj].argtype == 0 && fia == -1)
    {
      fii = ii;
      fia = jj;
    }
  }

  for (jj=0; jj < instance[ii].arguments; jj++)
  {
    int ci = instance[ii].argcode[jj].connector.instance;
    int ca = instance[ii].argcode[jj].connector.argument;

    if (ci == -1)
    {
      if (instance[ii].argcode[jj].argtype == ARG_STREAM_IN ||
          instance[ii].argcode[jj].argtype == ARG_STREAM_OUT)
      {
        /* Make new variable to interact with the outside world */
        code newid = makeid(‘oa‘, ca);

        @declare(addrpointer(instance[ii].argcode[jj].datatype),
                 newid);

        for (kk=1; kk < 4; kk++)
          skelcode[ii].split[kk] =
            replace(instance[ii].argcode[jj].id,
                    ‘($@newid)[pbp_ii]‘, skelcode[ii].split[kk]);

        instance[ii].argcode[jj].id = newid;
      }

    }
    else if (instance[ii].argcode[jj].argtype == ARG_STREAM_IN)
    {
      /* Connect by referencing output variable */
      for (kk=1; kk < 4; kk++)
        skelcode[ii].split[kk] =
          replace(instance[ii].argcode[jj].id,
                  instance[ci].argcode[ca].id,
                  skelcode[ii].split[kk]);

      instance[ii].argcode[jj].id = instance[ci].argcode[ca].id;
    }
  }
}
```

```
stride =
  scuBufferGetStride(args[instance[fii].argcode[fia].
                          connector.argument].buffer);
pbp_hpixels = stride;

/* Preprologue (initialize variables) */
for (ii=0; ii < instances; ii++)
  @skelcode[ii].split[PREPROLOGUE];

propEnvReset();
propEnv("operation", operation);

for (ii=0; ii < instances; ii++)
  for (jj=0; jj < instance[ii].arguments; jj++)
  {
    int ci = instance[ii].argcode[jj].connector.instance;
    int ca = instance[ii].argcode[jj].connector.argument;

    if (ci == -1)
    {
      propEnv("argument", @idtostring(makeid(`id`, ca)));
        propSeti("id", ca);
        propEnv("target", target);
          propSeti("buffer", instance[ii].argcode[jj].state.buffer);
          propSeti("state", instance[ii].argcode[jj].state.state);
          propSeti("delay", instance[ii].argcode[jj].state.delay);
          propSeti("elementsize",
                   sizeof(instance[ii].argcode[jj].datatype));

          if (instance[ii].argcode[jj].argtype == ARG_STREAM_IN ||
              instance[ii].argcode[jj].argtype == ARG_STREAM_OUT ||
              instance[ii].argcode[jj].argtype == ARG_LOOKUP)
            propSeti("chunksize", stride);
          else
            propSeti("chunksize", 1);


          if (instance[ii].argcode[jj].argtype == ARG_SCALAR_IN ||
              instance[ii].argcode[jj].argtype == ARG_SCALAR_OUT ||
              instance[ii].argcode[jj].argtype == ARG_LOOKUP)
            propSets("timedist", "bulk");
          else
            propSets("timedist", "regular");
        propEnvBack();
      propEnvBack();
    }
  }

/* Prologue */
for (ii=0; ii < instances; ii++)
```

```
  @skelcode[ii].split[PROLOGUE];

/* Actual processing */
while (pbp_hpixels =
         scuBufferPeek(args[instance[fii].argcode[fia].
                            connector.argument].buffer,
                       args[instance[fii].argcode[fia].
                         connector.argument].reader,
                       (void**)&(@instance[fii].argcode[fia].id),
                       stride *
                         sizeof(instance[fii].argcode[fia].datatype),
                       SCU_BLOCK_ALL))
{
  pbp_hpixels =
    pbp_hpixels / sizeof(instance[fii].argcode[fia].datatype);

  for (ii=0; ii < instances; ii++)
    for (jj=0; jj < instance[ii].arguments; jj++)
    {
      argcode_t *ac = &instance[ii].argcode[jj];

      if ((ii != fii || jj != fia) && (*ac).argtype == 0 &&
          (*ac).connector.instance == -1)
        scuBufferPeek(args[(*ac).connector.argument].buffer,
                      args[(*ac).connector.argument].reader,
                      (void**)&(@instance[ii].argcode[jj].id),
                      pbp_hpixels *
                        sizeof(instance[ii].argcode[jj].datatype),
                      SCU_BLOCK_ALL);
      else if (instance[ii].argcode[jj].argtype == ARG_STREAM_OUT &&
               (*ac).connector.instance == -1)
        scuBufferAllocate(args[(*ac).connector.argument].buffer,
                          (void**)&(@instance[ii].argcode[jj].id),
                          pbp_hpixels *
                            sizeof(instance[ii].argcode[jj].datatype),
                          SCU_BLOCK_ALL);
    }

  for (pbp_ii=0; pbp_ii < pbp_hpixels; pbp_ii++)
    for (ii=0; ii < instances; ii++)
      @skelcode[ii].split[REPEAT];

  for (ii=0; ii < instances; ii++)
    for (jj=0; jj < instance[ii].arguments; jj++)
    {
      argcode_t *ac = &instance[ii].argcode[jj];

      if (instance[ii].argcode[jj].argtype == ARG_STREAM_IN &&
          (*ac).connector.instance == -1)
        scuBufferReleasePeeked(args[(*ac).connector.argument].buffer,
                               args[(*ac).connector.argument].reader,
```

```
                                    pbp_hpixels *
                                      sizeof(instance[ii].argcode[jj].
                                                datatype));
        else if (instance[ii].argcode[jj].argtype == ARG_STREAM_OUT &&
                 (*ac).connector.instance == -1)
          scuBufferReleaseAllocated(args[(*ac).connector.argument].buffer,
                                    pbp_hpixels *
                                      sizeof(instance[ii].argcode[jj].
                                               datatype));
      }
  }

  /* Epilogue (release buffers) */
  for (ii=0; ii < instances; ii++)
    @skelcode[ii].split[EPILOGUE];
}
```

Summary

Automated Design of Application-Specific Smart Camera Architectures

Wouter Caarls

Parallel heterogeneous multiprocessor systems are often shunned in embedded system design, not only because of their design complexity but because of the programming burden. Programs for such systems are *architecture-dependent*: the application developer needs architecture-specific knowledge to implement his algorithms, as each processor has its own characteristics and programming language. He will therefore often stick to the architectures he knows best instead of looking for the best one. This leads to suboptimal solutions, and costly redesign efforts if the chosen architecture later proves to be insufficient. We demonstrate that there is no single best architecture or programming language that can release us from this plight.

Our solution to this problem uses a programming model based on the concept of *architecture independence through algorithm dependence*. By limiting the expressiveness of a programming language to just those concepts needed to implement a given class of algorithms, it may be compiled to a variety of different (parallel) processor architectures. In particular, we limit the access patterns an algorithm may employ using *algorithmic skeletons*. Using different languages – or *interfaces* – to implement different algorithms is more natural than using architecture-dependent languages, because a programmer should be concerned with algorithms, not with processors. A language that is tailored to a specific class of algorithms will allow those algorithms to be expressed more naturally than using a general-purpose or processor-specific language.

We have designed a meta-programming system to implement translators for such *algorithm-specific languages* (ASLs). A translator for an ASL has three distinct tasks. First, it must implement the access pattern that is part of the skeleton, such as a row-major iteration over image pixels. Second, it must rewrite the algorithm-specific interface to that used by the run time system – for example, changing relative array indexing to absolute indices. Finally, it must translate the algorithm (also called the *kernel*) into a language that is understood by the target processor. PEPCI, a new meta-programming language and tool for implementing ASLs, facilitates these tasks by allowing the pattern to be implemented directly in the target language while still providing sophisticated code transformation tools.

The operations resulting from the translation are shown to perform within 20% of their handcrafted counterparts, in which the latter can be considered optimal.

An application consists of multiple algorithms, each written in its own language dialect. In image processing applications, these algorithms often work in successive steps on a series of images. This suggests the view of an application as a directed graph of tasks working on streams of data. This is the view taken in *stream programming*, and we present our implementation of that concept (based on *remote procedure call (RPC)*). Using *futures*, RPC allows the application's task graph to be built under imperative program control, thereby allowing it to be data-dependent. Such data dependencies are important in applications that need to operate in dynamic environments, or in dynamic roles. An automated, dynamic mapping of the task graph to an *architecture model* maintains the architecture independence of the program. We demonstrate that this mapping creates an effective multiprocessor program, but that most of the benefits of using a heterogeneous system must be realized by parallelizing the operations themselves.

Finally, we use the architecture independence of programs written using our programming model to conduct an automated *design space exploration* of possible architectures, creating a *Pareto front* of optimal trade-offs between performance, area and power consumption. Each design point is simulated, using prebenchmarked operations to speed up the process. The simulation has an accuracy of around 10%. We use *multi-objective evolutionary optimization* to approximate the Pareto front, and show that this has good convergence and coverage. The practicality of the entire system of skeletonization, stream programming and design space exploration is demonstrated by implementing two case studies.

Samenvatting

Geautomatiseerd Ontwerp van Toepassingsspecifieke Intelligente Camera Architecturen

Wouter Caarls

Parallele heterogene multiprocessor systemen worden vaak vermeden bij het ontwerp van embedded systems, niet alleen omdat ze moeilijk te ontwerpen zijn, maar vooral vanwege problemen met de programmeerbaarheid. Programma's voor zulke systemen zijn *architectuur-afhankelijk*: de ontwikkelaar heeft architectuur-specifieke kennis nodig om zijn algoritmes te implementeren, omdat elke processor verschillende eigenschappen en een eigen programmeertaal heeft. Een ontwikkelaar zal daarom vaak kiezen voor architecturen waar hij bekend mee is, in plaats van te zoeken naar de beste architectuur. Dit resulteert in suboptimale oplossingen, en dure herontwerp trajecten als de gekozen architectuur later niet blijkt te voldoen. We laten zien dat er geen "beste" architectuur of programmeertaal bestaat om dit probleem te vermijden.

Onze oplossing maakt gebruik van een programmeermodel dat gekenmerkt wordt door *architectuur onafhankelijkheid door algoritme afhankelijkheid*. Door de expressiviteit van een programma te beperken tot alleen die concepten die nodig zijn om een bepaalde klasse van algoritmes te implementeren, kan het naar een keur van verschillende (parallele) processorarchitecturen worden gecompileerd. De belangrijkste beperking wordt opgelegd door de manier waarop geheugen kan worden benaderd te beperken met behulp van van *algoritmische skeletten*. Het gebruik van meerdere talen – of *interfaces* – om verschillende algoritmen te implementeren is natuurlijker dan het gebruik van architectuur-afhankelijke talen, omdat een programmeur zich bezig houdt met algoritmen, niet met architecturen. Een programmeertaal die is toegespitst op een bepaalde klasse van algoritmen is geschikter om die algoritmen in uit te drukken dan een algemene of processor-specifieke taal.

We hebben een metaprogrammeersysteem ontworpen om compilers voor zulke *algoritme-specifieke talen* (*algorithm-specific languages, ASLs*) te implementeren. Een compiler voor een ASL kent drie verschillende taken. Ten eerste moet het de geheugentoegangspatronen van het algoritmische skelet implementeren, bijvoorbeeld een iteratie over de pixels van een beeld. Ten tweede moet het de algoritme-specifieke interface herschrijven naar de interface die gebruikt wordt door het run-

time systeem, zoals het vervangen van relatieve pixel-adressering door absolute adressering. Ten slotte moet het het algoritme zelf (ook *kernel* genoemd) vertalen naar een taal die ondersteund wordt door de processor. PEPCI, een nieuwe metaprogrammeertaal en -systeem om ASLs te implementeren vereenvoudigt deze taken doordat het patroon direct in the doeltaal kan worden geïmlementeerd, terwijl het toch geavanceerde mogelijkheden biedt voor het herschrijven van programma's. We laten zien dat de operaties die het resultaat zijn van de vertaling binnen 20% van (als optimaal te beschouwen) handgeschreven code presteren.

Een applicatie bestaat uit meerdere algoritmen, elk uitgedrukt in zijn eigen taal. In beeldverwerkingsapplicaties zijn de algoritmen vaak opeenvolgende stappen die op een serie beelden worden toegepast. We kunnen zo'n applicatie zien als een gerichte graaf van taken die op een stroom van data werken. Dit is het gezichtspunt dat wordt ingenomen in *stream programming*, en we presenteren onze implementatie van dat concept (gebaseerd op *remote procedure call (RPC)*). Door gebruik te maken van *opties* kunnen met RPC taakgrafen op een imperatieve manier geconstrueerd worden, waarbij de structuur data-afhankelijk kan zijn. Zulke data afhankelijkheden zijn belangrijk bij toepassing in dynamische omgevingen, of in verschillende rollen. De architectuuronafhankelijkheid wordt bewaard door een geautomatiseerde, dynamische toewijzing van de taakgraaf aan de processoren in een *architectuurmodel*. We laten zien dat deze toewijzing een goed multiprocessorprogramma oplevert, maar dat de meeste winst van een heterogene architectuur moet worden behaald door het paralleliseren van de operaties zelf.

Ten slotte gebruiken we de architectuuronafhankelijkheid van programma's die voor ons programmeermodel geschreven zijn om automatisch de *ontwerpruimte* te doorzoeken op geschikte architecturen. Dat levert een *Pareto front* op van optimale afwegingen tussen snelheid, chipoppervlak en energieverbruik. Elk punt in de ontwerpruimte wordt gesimuleerd, waarbij we gebruikmaken van aparte metingen voor elke operatie om de simulatie te versnellen. De simulatie heeft een nauwkeurigheid van ongeveer 10%. We gebruiken *evolutionaire optimalisatie voor meerdere doelen* om het Pareto front te benaderen, en laten zien dat dit goede convergentie-eigenschappen heeft. De bruikbaarheid van het hele systeem van skeletonisatie, stream programming en doorzoeken van de ontwerpruimte wordt gedemonstreerd door twee case studies te implementeren.

Acknowledgements

When Pieter Jonker first approached me for the SmartCam project, I was immediately enthralled. Parallel computing had been one of my favorite subjects, and the opportunity to design my own processor architecture was too good to pass up. I would like to thank him for this opportunity, his support throughout the entire project, and for giving me the freedom to structure the work as I thought best.

I would also like to thank Henk Corporaal as the leader of the SmartCam project, for his feedback on my papers. Without his outside but knowledgeable opinion, they would have been much less readable. Our other partners in the project, Richard Kleihorst, Harry Broers, and Hamed Fatemi also provided valuable feedback during our regular meetings.

Thanks also to my promotor Lucas van Vliet, who diligently read and corrected the thesis manuscript during the last stage. It was no concern that the subject was outside his usual field, for he is interested in – and knowledgeable and vocal about – everything around him.

Many thanks go to the other Ph.D. students and staff in PH/QI, with whom it was a pleasure to share the coffee room, with its many scientific and unscientific discussions. I would especially like to thank my roommates Frank, Vincent, and Kees, for sharing their problems and listening to mine. Also, thanks to Cris, Dick, and Mike, for making me feel welcome and not hitting me in SoF2, and to Bernd and Frank for the many nights of playing board games.

I would also like to mention the M.Sc. students who occupied the Robocup room over the years, some of whom I have had the pleasure of supervising, and others whose company I enjoyed during tournaments: Bram, Corné, Erik, Floris, Jan, Jan-Willem, Jev, Krijn, Martijn, and Paul.

Finally, I would like to thank my parents and brother for supporting and encouraging me, and without whose warmth and help – both trivially and non-trivially – this thesis would never have been.

Curriculum Vitae

Wouter Caarls was born in Amsterdam on the 15th of November, 1978. He obtained his VWO diploma at the Teylingen College, in Noordwijkerhout. In 1997 he went to Amsterdam to study Artificial Intelligence at the University of Amsterdam where he received his Master's degree in 2002, with honor. The subject of his Master's thesis was "Genetic Algorithm Visualisation". The research was conducted at the Section Computational Science under supervision of Jaap Kaandorp.

In July 2002 he joined the Quantitative Imaging group, department of Imaging Science and Technology, at the Delft University of Technology, where he worked on the PROGRESS (PROGram for Research on Embedded Systems & Software) project EES.5411 (SmartCam: Devices for Embedded Intelligent Cameras). The results of the work are presented in this thesis. He spent a month working at Philips Research Laboratories, Eindhoven, and NEC Corporation, Kawasaki, Japan, in November 2002 and May 2005, respectively.

In August 2007 he joined the Laboratory of Cellular Dynamics at the Max Planck institute for Biophysical Chemistry in Göttingen, Germany.