UNIVERSITY OF CALIFORNIA, SAN DIEGO

Genetic Compilation for Tiled Microprocessors

A thesis submitted in partial satisfaction of the
requirements for the degree Master of Science
in
Computer Science

by

Jin Seok Lee

Committee in charge:

Professor Michael B. Taylor, Chair
Professor Steven Swanson
Professor Yoav Freund

2007

The thesis of Jin Seok Lee is approved:

Chair

University of California, San Diego

2007

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First, I would like to thank to my research advisor Professor Dr. Michael Bedford Taylor whose perceptive criticism, insight into tiled architectures and willing assistance helped me bring about successful research. I would also like to thank Dr. Steven Swanson and Dr. Yoav Freund as my committee members. Also thanks to my office mates (Donghwan Jeon, Ikkjin Ahn and Hyojin Sung) for helping me understand tiled architectures. In addition I am grateful for the help of my friends at Onnuri.

Finally, thanks to my family and friends for encouraging me to continue my studies in San Diego.

This work was supported by the Korea Science and Engineering Foundation Grant (KRF–2005–215–D00289).

ABSTRACT OF THE THESIS

Genetic Compilation for Tiled Microprocessors

by

Jin Seok Lee
Master of Science in Computer Science
University of California, San Diego, 2007
Professor Michael B. Taylor, Chair

Recent microprocessor designers have turned to large-scale parallelism and multicore processors as the means of continuing Moore's Law. *Tiled multicore* processors, one such class of multicore processors, offer extremely low latency communication over an on-chip scalar operand network (SON). Although academic projects such as Raw and TRIPS have demonstrated that tiled multicore processors are implementable, managing the complexity of optimizing compilers for these distributed architectures has become a serious issue. These compilers must simultaneously optimize across a variety of inter-related NP-complete criteria in order to generate optimized code.

Generating custom compiler heuristics for these NP-complete problems requires high skill levels and is both time-consuming and prone to over-simplification of the emerging factors that contribute to sub-optimal parallel speedup. Furthermore, the implementation effort of these heuristics makes it almost impossible to adjust them in order to evaluate tradeoffs in tiled microprocessor design.

This thesis presents a complete compiler backend that generates parallel code for tiled microprocessors. It addresses complexity issues by separating the concerns of correctness and optimization. The optimization component uses standard machine learning algorithms (genetic programming), while the correctness component ensures that valid code is generated regardless of the input from the machine learning algo-

rithm. The evaluation measures the compiler's ability to tune the placement of memory objects across tiles; in several cases it is able to perform placement better than a graduate student. Furthermore, it does this with no understanding, beyond what is necessary to generate correct code, of the particular target architecture (Raw).

1

Introduction

Until recently, modern microprocessors have focused on clock rate as the means of exploiting improvements in silicon manufacturing due to Moore's Law [13] and CMOS scaling. This approach has come up against complexity and scalability limits, including those due to wire delay, power, and logic delay. Recently, microprocessor designers have turned to large-scale parallelism and multicore processors, which integrate many processors onto one silicon die, as the means of continuing Moore's Law. These new multicore processors carry with them significant challenges in programmer productivity, as they frequently require explicit manual parallelization. One approach to addressing this issue is to create architectures for which compilers can automatically generate code. *Tiled multicore* processors are a subclass of multicore processors which facilitate automatic compilation by providing extremely low latency communication over an on-chip scalar operand network (SON) [18, 19]. RAW [17, 20], Wavescalar [16] and TRIPS [4] are three examples of academic tiled architectures. Tiled architectures have integrated more simple and identical tiles into a chip to address the limits. These projects have shown that tiled architectures have the ability to run a variety of codes in parallel. In this chapter, we introduce the overall structure of typical tiled architectures, the major concepts of compilers for tiled architectures, and finally the motivation for this thesis.

1.A Tiled architectures

Tiled architectures are designed to efficiently run programs in parallel across a large silicon area. In order to exploit parallelism, tiled architectures are constructed in a physically scalable and simple way. A tiled microprocessor consists of an array of identical tiles (shown Figure 1.1). Each tile contains a compute portion and a networking portion. The compute portion contains a compute pipeline (including ALU, and FPU), an instruction cache, and a data cache. The communication components contains programmable routers (also called switches) and network wires that connect the tile to its neighbors and off-chip. The tiles are connected by a variety of point-to-point, pipelined, on-chip networks which facilitates low-latency communication among tiles and among tiles and off-chip devices, via the I/O ports. This construction provides a variety of benefits for overcoming scalability limitations in microprocessor design. Figure 1.2 shows a die photograph of the MIT Raw microprocessor, which contains 16 such tiles. The number of tiles is expected to double with each generation of Moore's Law (e.g., 1024 tiles at the 22 nm process node).

One of the important features of tiled architectures is that they are naturally suitable for exploiting Instruction Level Parallelism (ILP) across multiple tiles, while a monolithic superscalar processor requires complex hardware resources to do ILP. ILP can be found and exploited automatically in sequential programs written in C or C++. The compiler, targeted specially for tiled architectures, parallelizes the program across the tiles. A separate instruction stream runs on each tile with its own Program Counter (PC). Although independent programs may run simultaneously on different tiles in the same way they do on multicore processors, tiled microprocessors also allow the tiles to cooperate with their neighbors via static or dynamic interconnection network. After individual instructions are mapped into tiles, each tile has to run instructions assigned to the tile and also orchestrate instruction execution with neighboring tiles via the interconnection networks, transporting scalar values from producer tiles to the appropriate consumer tiles. Such a scheme is called a *Scalar Operand Network* or

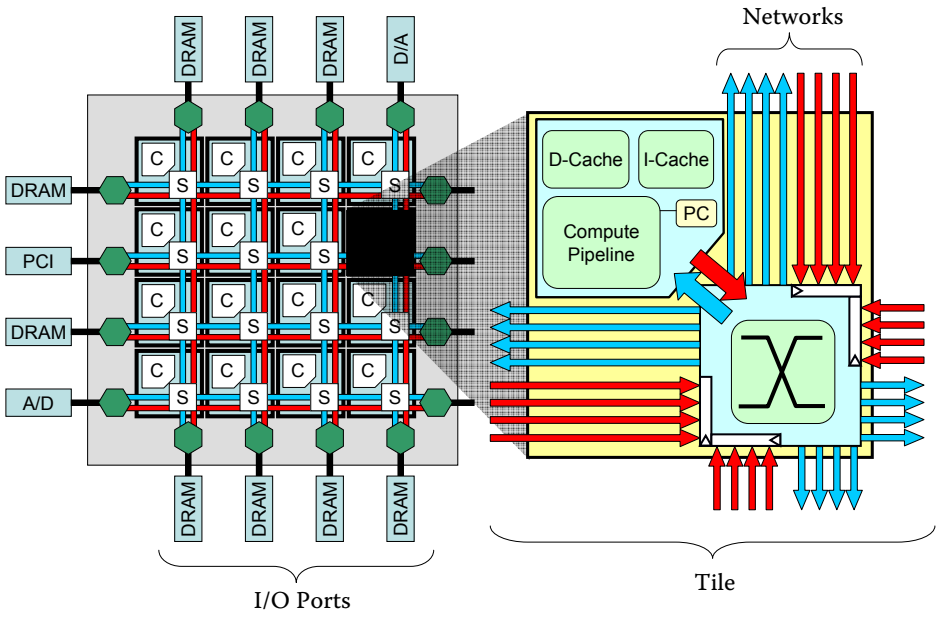


Figure 1.1: A tiled architecture. Used with permission from Prof. Michael Taylor.

SON [18].

Tiled architectures keep hardware as concise and simple as possible in order to provide better scalability for computation and on-chip memory, in contrast to complex modern super-scalar microprocessors. To efficiently utilize these scalable arrays of resources, intelligent compilers must be constructed to orchestrate the mapping of computations to architectural resources. As a result, many aspects of the basic underlying architecture associated with program execution are completely exposed to compiler and runtime system. The compiler is responsible for assigning and scheduling program instructions in order to exploit ILP. Program execution is almost completely controlled by the compiler, and thus the performance of programs depends on efficient resource assignment. The correctness of program execution is also determined by the compiler.

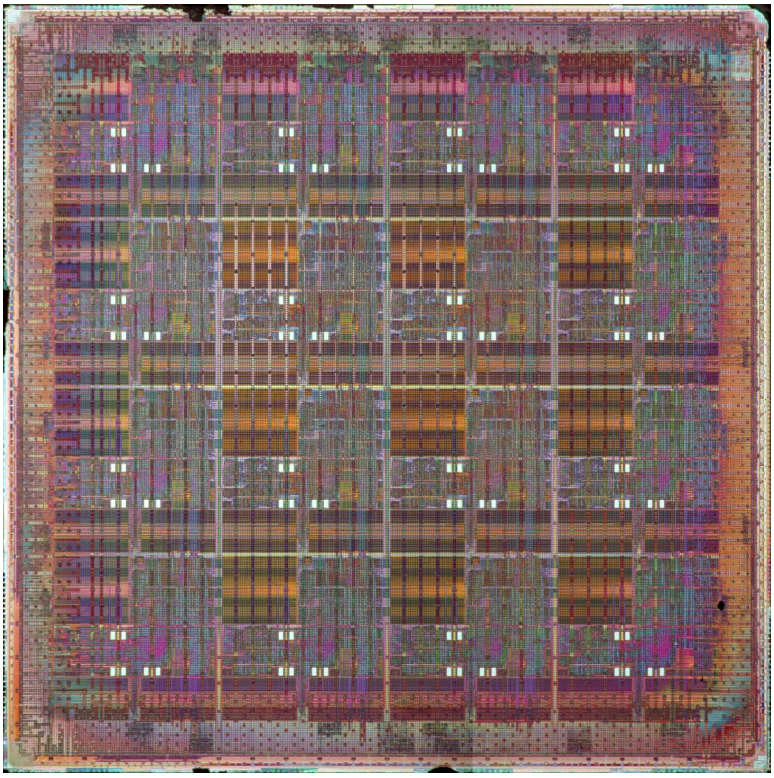


Figure 1.2: The photograph of the silicon die of the 16-tile MIT Raw Tiled Microprocessor. The 16 tiles are clearly visible. Used with permission from Prof. Michael Taylor.

1.B Compilation for Tiled Architectures

As discussed in the previous section, the hardware structure of tiled architectures is much simpler than traditional single-core wide-issue microprocessors. It ensures high clock rates and the availability of many execution resources on-chip. Instead of employing complex hardware on a chip, tiled microprocessors rely on compilation techniques which attempt to assign elements in the source code to hardware resources through the Instruction Set Architecture (ISA) abstraction. Researchers at MIT created a compiler for tiled architecture named RawCC [2, 3, 10]. However, this compiler is tightly coupled to the Raw tiled microprocessor. To handle the class of more general tiled architectures, we implemented a general compiler to compile a single stream program into multiple streams and exploited ILP on tiled architecture.

In lieu of supporting complex run-time hardware, the compiler plays a dominant role in determining a static path of program execution in compile-time with affordable algorithms. Most resources used in a program's execution are controlled by a sequence of phases (or *passes*) in a back-end of a compiler. This paper is devoted to describing the design, implementation and performance of the back-end of the compiler. The overall back-end flow of the compiler is described in Figure 2.1. We shall examine the phases in Chapter 2 in more detail.

1.C Motivation

Central to this thesis are two ideas: first, a new method of organizing compilers so that the concerns of correctness and optimization are separated, reducing the complexity of compiler implementation; and second, the application of brute-force machine learning algorithms to replace the costly and time-consuming construction of heuristic optimization functions for compilers for sophisticated distributed architectures. To explore these ideas, we construct a complete backend compiler for next generation tiled architectures, and measure the benefit of applying machine learning

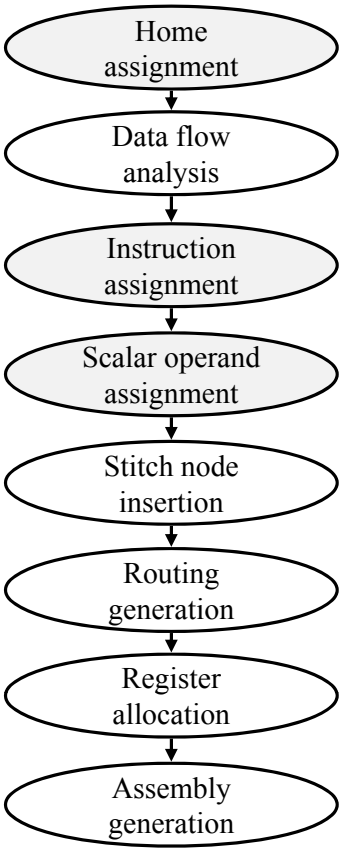


Figure 1.3: Back-end passes of a compiler for tiled architectures

techniques (genetic algorithms [8]) to optimize memory object placement, which has significant downstream effects on compiled code performance.

This new backend infrastructure is designed to target the general class of tiled architectures, and to support new emerging tiled architectures. Until now, tiled architectures have been examined and implemented mostly in academia. Compilers for those architectures have targeted specific architectures and even specific prototypes. As a result, targeting new tiled architecture typically will require an almost from-scratch rewrite of the existing compilers, tuned with new heuristics to address particular issues of the individual architecture. Compiler writers have to spend much of their time and energy in re-implementing new infrastructures, delaying work on new compiler algorithms and new architectural features. In the interests of reducing this effort, we propose a general compiler infrastructure for tiledarchitecture.

A compiler for a tiled architecture has to find approximations to a cascade of NP-hard problems in order to generate optimized code. Compiler developers must often struggle to manage the complexity of finding and implementing efficient approximation algorithms. Although our compiler addresses many of the standard phases of tiled architecture compilation, to keep the scope of this thesis maintainable, we will focus on the compiler's ability to find efficient memory objects placements, which is central to the exploitation of parallelism in ILP codes. This both demonstrates the feasibility of our machine-learning approach, but also extends the current state of the art in tiled compiler research. Most of the extant research in tiled architecture compilation tends to center around instruction placement and scheduling [10, 12]. Up until now, few have attempted to address the issue of memory placement in tiled architectures. The issue of memory placement is challenging because it simultaneously affects how much memory parallelism is available in the application, while at the same time, it heavily influences the ability of the architecture to place non-memory instructions in ways that reduce network latency and congestion.

The remainder of the thesis is organized as follows: Chapter 2 presents the

overview of the compiler design and implementation. In Chapter 3, we evaluate the compiler in the context of using genetic algorithms to optimize memory placement. Finally, we conclude the discussion in Chapter 4, and overview the issue of memory object migration in Appendix D.

2

Compilation phases

The compiler is organized as a series of compilation phases for mapping programs to a set of tiled resources as efficiently as possible. We focus on the back-end of the compiler, as the back-end component changes the most when tiled architectures are targeted. This chapter explains in detail how each phase in the compiler works and how it ensures the correctness of programs. Figure 2.1 shows the overall compiler flow.

Shaded circles indicate subproblems, which are often NP-hard, that need to be solved. To simplify the effort of constructing the compiler, we separated the implementation of code correctness and optimization. In the first version of the compiler, the optimization functions were implemented with simple calls to the *random* function. This method of implementation helped us verify that the generated code was correct regardless of the input of the machine learning optimization functions. Later, we employed simple heuristics and/or machine learning for these optimization functions.

2.A Input

The inputs for the compiler are Hardware Description XML and Source Code XML. Hardware description XML is used to describe the hardware configuration for the target tiled architecture. The compiler assigns resources and generates code based

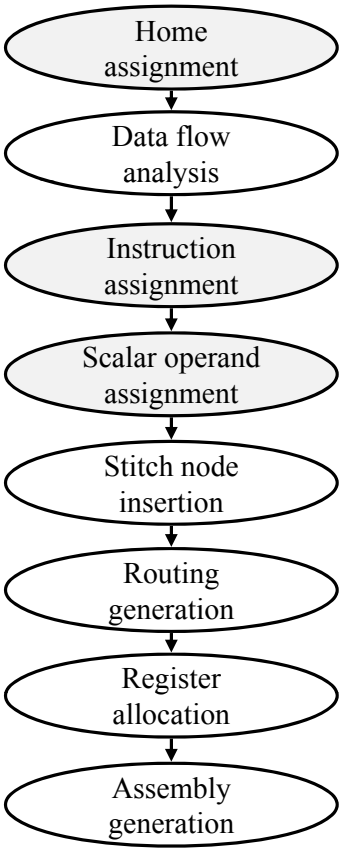


Figure 2.1: Back-end passes of a compiler for tiled architectures

```
<root>
        <RegisterNum>32</RegisterNum>
        <TileLoc>0</TileLoc>
        <TileLoc>1</TileLoc>
        <Latency>
                <LD>2</LD>
                <ST>1</ST>
                <ADD>1</ADD>
                <SUB>1</SUB>
                <MUL>12</MUL>
                <DIV>35</DIV>
                <FADD>2</FADD>
                <FSUB>2</FSUB>
                <FMUL>4</FMUL>
                <FDIV>12</FDIV>
        </Latency>
</root>
```

Figure 2.2: An example of hardware description XML

on the characteristics of the XML. Figure 2.2 shows an example of hardware description XML for the Raw microprocessor.

The other input file for the compiler is the source code XML file. The front-end of the compiler translates program source code written in MATLAB, C or C++ into this XML format. The XML represents a control flow graph (CFG) of the source code. It can include many control flow graph nodes, and each CFG node identifies the locations of its successor and predecessor nodes. The graph structure is provided by the front-end of the compiler through control flow analysis.

While most instructions in a CFG are similar to typical assembly instructions, some high-level operations are defined in the Intermediate Representation (IR). As this compiler has been designed for general tiled architecture, some features related to specific target machines are abstracted out through the IR. Four IR instructions are recognizable in the IR translation phase of the compiler - def, return, func and inargs. The representation of each IR is listed below.

1. *def* IR allocates space for memory objects in a stack. The “size” element in *def* decides the number of bytes to be allocated.

2. *return* IR returns a scalar value to a caller by default. It is also able to return

```
<root>
  <CFG>
    <CFGNode>
      <NodeID>0</NodeID>
      <Succ1>0_0</Succ1>
      <InstructionNode>
        <Label>foo</Label>
      </InstructionNode>
      <InstructionNode>
        <OPCode>def</OPCode>
        <Scalar1>a0</Scalar1>
        <Non_Scalar>8</Non_Scalar>
        <Non_Scalar>8</Non_Scalar>
      </InstructionNode>
      <InstructionNode>
        <OPCode>def</OPCode>
        <Scalar1>a1</Scalar1>
        <Non_Scalar>8</Non_Scalar>
        <Non_Scalar>8</Non_Scalar>
      </InstructionNode>
```

Figure 2.3: An example of source code

multiple values, using a stack.

3. *func* IR calls a function. It has a target address of the function and the argument list.

4. *inargs* IR represents an argument list of a function.

2.B Home assignment

Home assignment assigns tile locations to memory objects. The *home tile* refers to a single tile location in which a memory object resides. If we assume the most general case, that the target tiled architecture is cache-incoherent, the memory object in each procedure must belong to a specified tile called the *home tile* [10]. Otherwise, the correctness of a program is compromised. In this version of the compiler, home assignments persist throughout the lifetime of the program. `Load` and `store` instructions that access a particular memory object are assigned to the same home tile as the corresponding memory object.

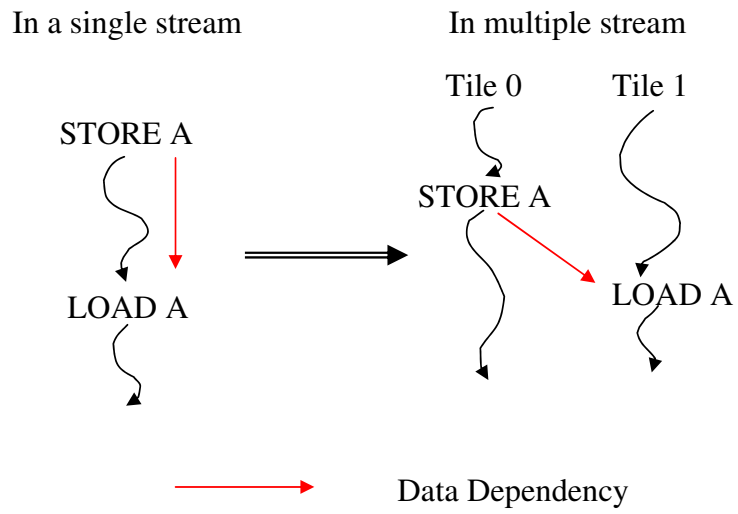


Figure 2.4: An example of incorrectness and incoherence memory

Home assignment is necessary to handle problems of incorrectness and incoherence. The term 'incorrectness' refers to memory objects and `load` or `store` instructions, which are not properly synchronized. The term 'incoherence' refers to the case where a tile's instruction stream accesses out-of-date versions of data even when synchronization is correct. Let's take a look at incorrectness in Figure 2.4.

A single stream code is transformed into multiple streams through this compiler. In Figure 2.4, tile 0 executes a `store` on `A` and tile 1 carries out a `load` on the same `A` as tile 0. Because each tile in a tiled architecture is only loosely coupled with the others during execution, tile 0 may execute `load A` before tile 1 executes the `store A`, which results in incorrect behavior.

In other words, a tile does not know if an instruction with dependent scalars on another tile has or will produce the corresponding values or not.

Tiled architectures with incoherent memory systems also pose problems of 'inconsistency'. Suppose the order of execution is correct in Figure 2.4. In some cases, Tile 1 may not see the correct value from tile 0 at all times because of the effects of caching. If `A` on tile 0 is not written back to memory before the execution of the

`load` on tile 1, `A` on tile 1 is from out-of-date memory, and is not equivalent to the `A` manipulated by `store A` on tile 0. Therefore, the compiler supports the concept of "Home" to tackle cache-incoherence in a back-end of the compiler.

To verify the correctness of the compiler implementation, we initially used the *random* function in performing home assignment. Memory objects were assigned to arbitrary tiles. As long as instructions such as `load` and `store` follow the rule that the instructions and referenced memory objects in them are in the same tile, a program is guaranteed to execute correctly. This is because `load` and `store` instructions in other tiles do not manipulate those memory objects, and thus cannot create coherency or synchronization issues.

For optimization of assigning memory objects, we have implemented home assignment using a genetic algorithm. It generates faster executable codes as generations elapse. The efficacy of the algorithm is evaluated in chapter 3 in detail.

2.C Data flow analysis

In this phase, we construct conventional 'def-use' chains [1] within a CFG node. This analysis identifies the data dependency relationships that exist between instructions that define scalars and instructions that use them. The dependency information becomes a basis of instruction assignment, scalar assignment and the generation of instructions is described in the following sections.

2.D Instruction assignment

This phase assigns each instruction within a CFG node to a specific tile and is quite important to quality of performance. If they are assigned in an inefficient way, data dependency between instructions across tiles generates redundant routing instructions.

To ensure correctness, the first version of the compiler for tiled architecture

assigned instructions across tiles in conjunction with a *random* function. To demonstrate the correctness of this phase, we should show that data transportation in a single tile and in multi tile is done correctly. First, if data dependency exists between two instructions within a tile, 'def-chain' is used as a method to transfer correct data. Second, if data dependency exists across tiles, routing instructions are responsible for transporting correct data. Therefore, this phase ensures that assignment is executed correctly whichever method is used.

For optimizing the execution of this phase, heuristics [11], [14] or machine learning algorithms [5], [9], [15] assign instructions in compile-time. We have developed an instruction assignment algorithm based on Unified Assign and Schedule (UAS) [14], which was originally created for Very Long Instruction Word (VLIW) machine. The main idea of this algorithm is to assign and schedule instructions simultaneously.

Though our compiler bears a resemblance to UAS in the assigning and scheduling functions, the compiler has different features such as interconnection network type, a space-time map and instruction assignment for tiled architectures.

The first difference is that the compiler for tiled architectures regards the network as a point-to-point network while UAS considers the network as inter-cluster buses. Secondly, the compiler makes different space-time map. Before instruction assignment, the compiler builds one two-dimensional space-time map (Figure 2.5). A row refers to one machine cycle on each tile, incrementing from top to bottom. A column represents a tile with two slots - one for a computation unit and the other for a switching unit. Third, the compiler assigns tile locations to instructions. A detailed scheme of instruction assignment is explained in appendix A.

2.E Scalar operand assignment

In this phase, the compiler decides where scalar values in a live-in list and a live-out list are located in a CFG node. In a monolithic microprocessor, each CFG

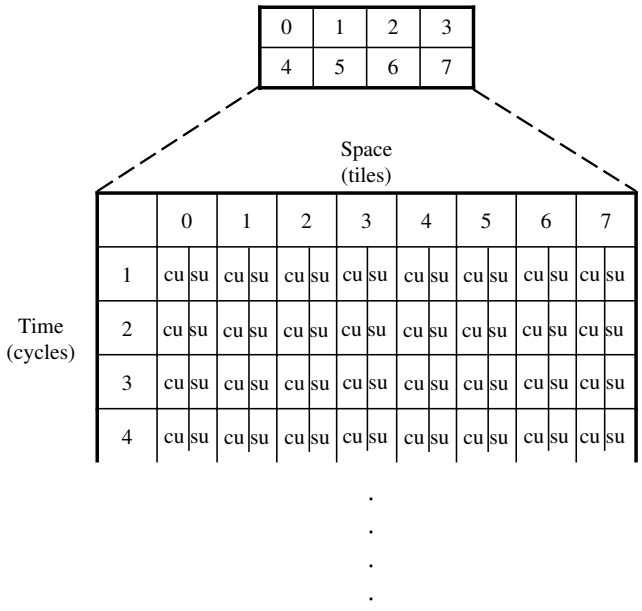


Figure 2.5: A space-time map of 8 tiles (cu : a slot for a computation unit, su : a slot for a switching unit)

node has a live-in list and a live-out list only for scalars. In tiled architectures, scalars as well as their locations are assigned to a live-in list of a tile and a live-out list because scalars in the two live lists are distributed across tiles. To facilitate this phase, we have built three sub-phases called *making a live-in list and a live-out list of each CFG node*, *assigning locations and registers to scalars in a live-in list and a live-out list of each CFG node* and *assigning scalar locations of instructions in each CFG node*.

. **Making a live-in list and a live-out list of each CFG node**: In this phase, the compiler runs a fixed-point algorithm [1], which continues to find scalars in a live-in list and a live-out list until there is no change in those lists.

  Figure 2.6, (b) shows an example of making a live-in list and a live-out list from the following instruction `add a,b,c` and `add d,e,f` on 2 tiles.

. **Assigning locations and registers to scalars in a live-in list and a live-out list of each CFG node**: The compiler assigns tile locations and virtual registers to scalars in a live-in list and a live-out list. Figure 2.6, (c) depicts an example

of assigning tile locations and virtual registers to scalars in a live-in list and a live-out list on 2 tiles.

. **Assigning locations and registers to scalars of instructions in each CFG node**: This phase assigns virtual registers and tile locations to scalars of instructions. The information is based on the routing instructions that have been generated. Receiving instructions recognize the register values that have been transferred. (d) of Figure 2.6 shows an example of assigning tile locations and virtual registers to scalars in every instruction within a CFG node on 2 tiles.

To ensure correctness, we applied the *random* function in scalar operand assignment. The function causes two problems - scalar inconsistency within a CFG node and scalar inconsistency between CFG nodes. (d) of Figure 2.6 shows two scalar inconsistency cases within a CFG node (live-in to instructions and instruction to live-out). These problems are addressed by generating routing instructions between them. Scalar inconsistency between CFG nodes are handled by *stitch* nodes. The concept of *Stitch* node is discussed in 2.F.

To optimize scalar operand assignment, we used a simple heuristic to assign the scalar operands. To create a live-in list, scalars in the live-in list copy the locations and register values of similar *use* scalars which is first appeared in a CFG node. To create a live-out list, scalars in a live-out list copy locations and register values of same *def* scalars which last appeared in a CFG node. If the CFG node does not have *def* or *use* scalars for live lists, the scalars retrieve tile location and register values from the closest CFG node.

2.F Stitch node insertion

A *Stitch* node ensures consistency between two adjacent nodes. In some cases, a live-out list of a predecessor node and a live-in list of a successor node do not match. Therefore, inconsistency occurs between two live lists in tile locations

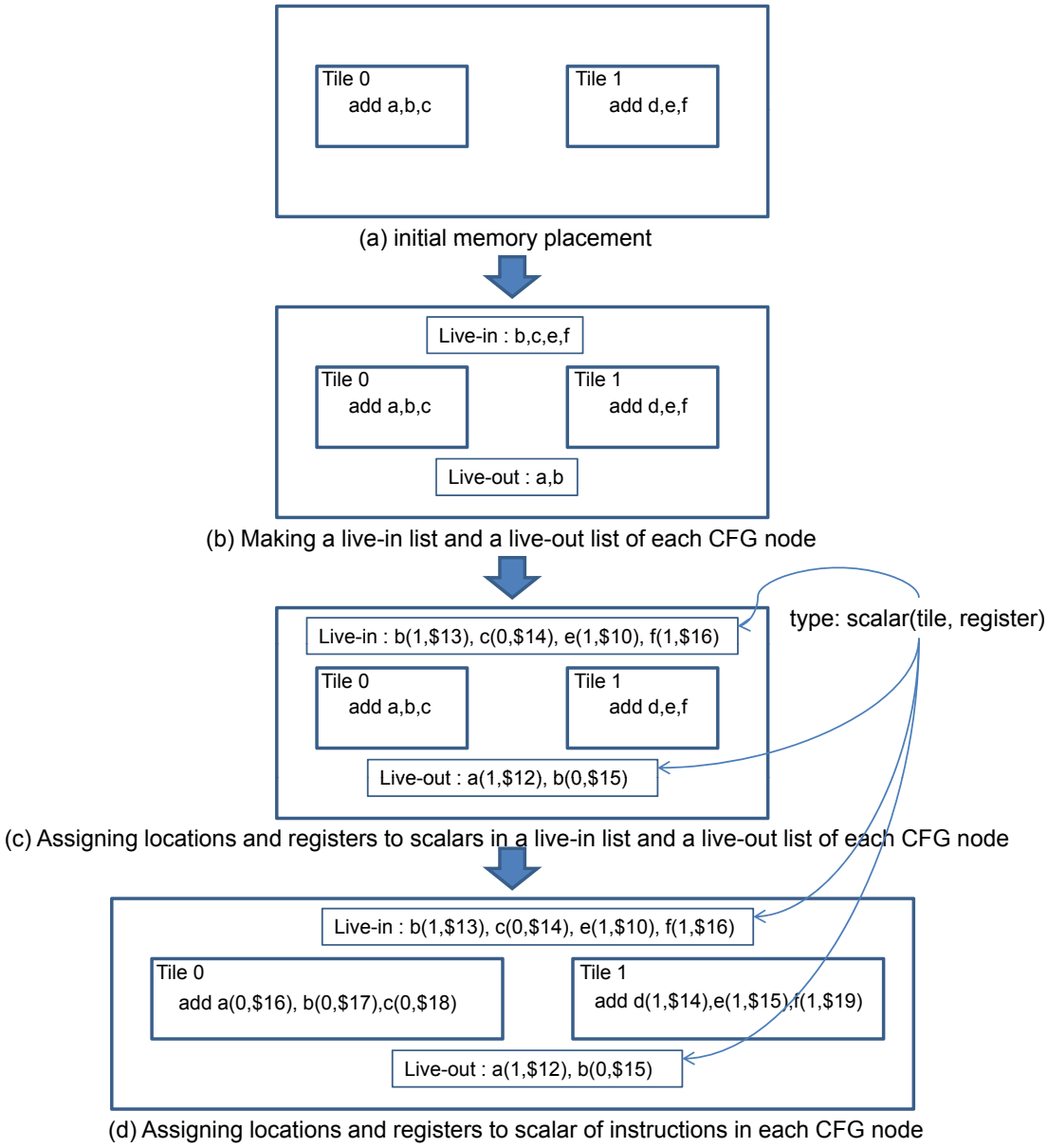


Figure 2.6: Scalar assignment flow example ((a, d) : def scalars, (b, c, e, f) : use scalars)

of scalars. The compiler deals with such a conflict, using *stitch* nodes. This phase handles possible mismatches of scalar operand assignment. The procedure is decided mechanically and is not greatly related to optimization. Appendix B shows how to insert a stitch node.

2.G Routing instruction generation

Based on 'def-use' chains, data dependency between two instructions across tiles is constructed. If the data dependency is across tiles, routing instructions are explicitly generated to transfer a scalar from one instruction, which produces a output, to another instruction, which consumes an input through dimensional order routing [6]. Appendix C shows how to generate routing instructions.

2.G.1 Optimization with ReDefScalars

Previous data analysis for routing generation was constructed with an assumption that only one tile remained alive. This type of analysis results in network overhead on multi tiles. The analysis could cause redundant codes to spread over on-chip networks. Consider Figure 2.7. Tile 0 produces `a` in `add` and tile 15 consumes it in instructions `sub` and `mul`. According to routing generation, routing instructions should be created for each scalar on tile 15.

This type of data analysis ignores new *def*s in routing generation although scalars routed over networks become new *def*s on destination tiles. *ReDefScalars* is able to those make those scalar values into new *def*s if no other instructions exist to define the scalar values between two instructions.

The compiler creates *ReDefScalars* for each tile. It contains scalar values and virtual registers. Whenever a scalar is routed to a destination tile in a CFG node, the tile writes a routed scalar and a virtual register in *ReDefScalars* of a tile. If the scalar is re-defined in other tiles, the scalar is deleted in *ReDefScalars*. Otherwise, the

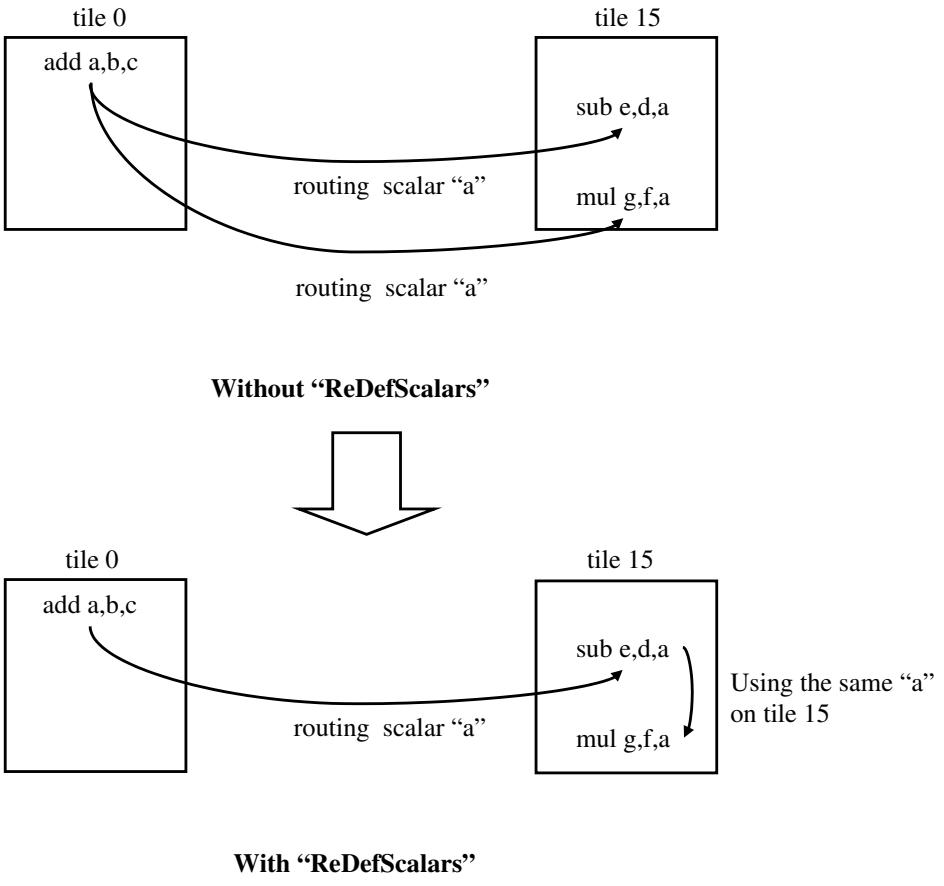


Figure 2.7: Examples of *without ReDefScalars* and *with ReDefScalars* ((a on tile 0, e, g on tile 15) : def scalars, (a, d, f on tile 15) : use scalars)

scalar survives in *ReDefScalars*. If the scalar in *ReDefScalars* of a tile exists, the scalar is reused in the tile without routing generation. In a case like 2.7, 8 cycles are saved, avoiding routing generation between tile 0 and tile 15.

2.H Register allocator

So far, all registers employed in every instruction are virtual registers or special registers such as the stack pointer or return address register. In register allocation, the compiler replaces virtual registers with real machine registers. To facilitate this process, we have adapted a standard coloring method used in [7] for multiple tiles. The standard coloring method is different from a coloring method in two ways.

1. Each tile owns its own register allocator as a slave. A single master register allocator supervises whole register allocators in every tile. The master allocator forces register allocators to run until all register allocations are not necessary anymore.

2. Stitch nodes are excluded in register allocation. All instructions in *stitch* nodes comprise routing instructions in which registers are defined in a switching unit and physically allocated in advance.

2.I Assembly generation

The last phase of the compiler enables the analysis of a single stream code and the creation of assembly '.S' files for multiple streams. '.S' files incorporate two assembly code sections (one for a computation unit and another for a switching unit). The instructions on a single stream are stored into each '.S' files based on their tile locations. Instructions for computation units are moved to a section of computation codes of the '.S' files and instructions regarding to the switch unit are placed in a section of switch codes.

3

Evaluation

The evaluation section of this thesis uses the complete compiler infrastructure, described in Chapter 2, to go from an XML description of the program to final object code for the Raw tiled microprocessor. In order to narrow the scope of inquiry, we focus on the evaluation of automatic genetic-algorithm-based memory placement. First, we present how experimental infrastructure is organized. Second, we evaluate the effectiveness of using a genetic algorithm to automatically perform memory placement.

3.A Experimental infrastructure

We implemented memory placement using a genetic algorithm. Figure 3.1 shows the execution flow of memory placement in genetic algorithm. First, it obtains a list of all memory objects in the source code. Then, it configures the parameters that control the genetic algorithm: the number of generations, the size of the population, the crossover rate and the mutation rate. After that, it creates an initial random population in which each memory object has been assigned a random tile number. To evaluate the efficiency of the members population (in this experiment, the tile assignment of all memory objects), a fitness value is generated through the measurement of the actual execution time. The genetic algorithm selects parent chromosomes which survive to

in next generation. Crossover and mutation are applied to the parents. Finally, a new generation is created with parental crossover and mutation. It continues to compare fitness values and to select the best chromosome until it generates the last generation.

A crucial factor in producing good object code from genetic algorithm is the accuracy of the fitness function. The more precise the fitness function, the more likely a genetic algorithm is able to discover solutions that optimize the many factors that contribute to performance. We employed the Raw cycle-accurate simulator (which is accurate to the exact cycle for over 250,000 lines of test code) in conjunction with the actual generated code and a sample data set in order to evaluate the fitness of a generated program. This approach has two advantages over using compiler estimates of program run-time. First, it does not burden the compiler-writer with the task of determining an appropriate algorithm for estimating execution time on a given algorithm. Understanding the first-, second-, and third- order effects of inter-related program parameters is a challenging task requiring not only high levels of skill but also substantial experience with a particular architecture. Relying on this deep level of understanding would impact our ability to quickly build compilers for new tiled architectures in order to explore the design space of tiled architectures. Second, using actual execution times reflects the most accurate possible estimate of runtime. This allows the genetic algorithm to take advantage of – or avoid – unexpected performance anomalies that result from the interaction of different factors in a program (e.g., register pressure, scheduling, and cache size).

Although this compiler is created to target the class of general tiled architectures, for accurate results, it remains to optimize for a specific machine. For this purpose, we choose the MIT Raw Tiled Microprocessor, for which a detailed cycle-accurate simulator exists. Evaluation of benchmarks is performed on a cycle-accurate simulator of the Raw machine. The compiler is made cognizant of only the most basic hardware characteristics (instruction types, number of tiles) of the Raw machine through the use of an XML description.

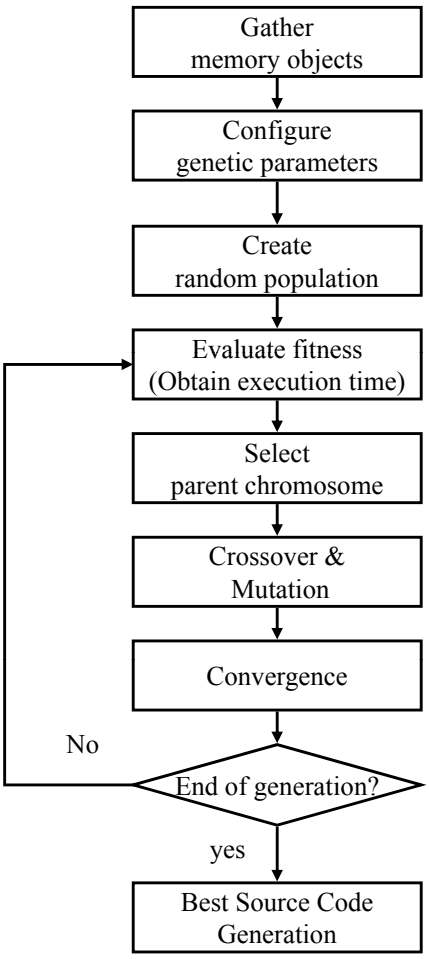


Figure 3.1: Genetic Algorithm in memory placement of the compiler

3.B High-level Source code transformations

The benchmarks we evaluate are convolution, dot-product and haar. A front-end part of this compiler analyzes input code and finds memory objects and scalars which are able to be parallelized. The compiler partitions memory objects into a number of memory objects that is on the same order as the number of tiles that is to be used to accelerate the program. Scalars, which control conditional branches, are also duplicated over all tiles.

Convolution is a discrete mathematical function which expresses the amount of overlap of one `kernel` when it passes one matrix A. Figure 3.2 shows how memory objects and scalars are transformed through a front-end of the compiler in convolution. Input memory object A is broken down into 16 memory objects. The compiler divides its width and height by 4. Each tile accesses the memory object by manipulated width and height. A scalar `sum` is also duplicated because it will be store in duplicated C. All control scalars such as width, height, uc and vc are replicated all tiles to eliminate transporting those scalars over tiles.

Dot-product takes two vector memory objects and multiplies each element with the same index in the two vectors (i.e., an elementwise vector multiplication). It returns the sum of these multiplied elements. Figure 3.3 shows that the compiler front end applies almost the same strategy as it does for dot-product. The input memory objects A and B are distributed across tiles. The scalar `sum` is also replicated. Dot-product returns the sum before the control is handed over to a callee.

Haar works by transforming an array of values into an array of average and differences-from-the-average. Figure 3.4 shows that Haar in this experiment is multi-layer Haar. First, a code runs the first Haar computation with original memory object input. Then, a front half of the first input becomes an input for next level of the Haar computation. The program continues the haar filtering until the layer number is one. Through the code transformation, the compiler partitions the input object for Haar into 16 memory objects, just like for the other algorithms above. A recursive call is

Table 3.1: Genetic algorithm parameters (population=the number of different genotypes per generation)

size of generations	100
size of populations	200
crossover rate	90%
mutation rate	10%

transformed into 4 iterations in a single function.

3.C Speedup of the genetic algorithm

In order to evaluate our automatic memory placement algorithm, we evaluate the benefit of a manually determined memory placement, which evenly distributes memory objects across tiles. The reasoning is that, if memory objects are evenly distributed over all tiles, this results in high levels of memory parallelism, and at least in theory, the best performance is gained with fully parallelized memory objects. We measure how much the genetic algorithm improves performance, compared to the manual memory placement. Parameters used from the genetic algorithm are described in Table 3.1.

Convolution uses two matrixes A (64×64), C (64×64) and a `kernel` (4×4). Figure 3.7 shows the interesting result that, for convolution, an unevenly-distributed memory object layout created by the genetic algorithm can outperform the evenly-distributed manual placement. Convolution has no relationship between matrixes. In Figure 3.2, convolution between A and `kernel` is stored in `sum`. After a `kernel` is applied to A, the `sum` is written in C. Therefore, evenly distributed manual placement has no effect on this evaluation. A result of evaluation depends on location of each scalar `sum` which sums convolution values from matrix A and delivers it to matrix C. The genetic algorithm has no problem with assigning memory objects at random.

```
A[0…width][0…height]
C[0…width][0…height]
for x:=0 to x<width do
   for y:=0 to y<height do
      sum: = 0
      for v:=-vc to v<=vc do
               for u:=-uc to u<=uc do
                     sum += A[x+v][y+u] * kernel[v+vc][u+uc]
      C[x][y]:=sum
```

```
A_0[0…width/4][0…height/4]  =  A[0…width/4][0…height/4]
A_1[0…width/4][0…height/4]  =  A[width/4…width*2/4][0…height/4]
        ⋮
A_15[0…width/4][0…height/4]  =  A[width*3/4…width][height*3/4…height]

C_0[0…width/4][0…height/4]  =  C[0…width/4][0…height/4]
C_1[0…width/4][0…height/4]  =  C[width/4…width*2/4][0…height/4]
        ⋮
C_15[0…width/4][0…height/4]  =  C[width*3/4…width][height*3/4…height]

for x:=0 to x<width/4  do
   for y:= 0 to y<height /4 do
      sum_0:=0 ; sum_1:=0 ; sum_2:=0 ; …  sum_14=0; sum_15=0
      for v:=-vc to v<=vc do
               for u:=-uc to u<=uc do
                     sum _0:= sum_0 + A_0[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _1:= sum_1 + A_1[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _2:= sum_2 + A_2[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _3:= sum_3 + A_3[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _4:= sum_4 + A_4[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _5:= sum_5 + A_5[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _6:= sum_6 + A_6[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _7:= sum_7 + A_7[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _8:= sum_8 + A_8[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _9:= sum_9 + A_9[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _10:= sum_10 + A_10[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _11:= sum_11 +  A_11[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _12:= sum_12 + A_12[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _13:= sum_13 + A_13[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _14:= sum_14 + A_14[x+v][y+u]  *  kernel[v+vc][u+uc]
                     sum _15:= sum_15 + A_15[x+v][y+u]  *  kernel[v+vc][u+uc]
         C_0[x][y]:=sum_0; C_1[x][y]:=sum_1; … C_14[x][y]:=sum_14; C_15[x][y]:=sum_15
```

Figure 3.2: convolution pseudocode after high-level transformation (width = width of input A, height = height of input A, vc : (width of kernel B)/2 , uc : (width of kernel B)/2))

```
A[0…width][0…height]
B[0…width][0…height]
sum:=0
for i:=0 to width do
        for  j:=0 to height do
                sum:=sum+A[i,j]*B[i,j]
return sum
```

```
A_0[0…width/4][0…height/4]  =  A[0…width/4][0…height/4]
A_1[0…width/4][0…height/4]  =  A[width/4…width*2/4][0…height/4]
   ⋮
A_15[0…width/4][0…height/4]  =  A[width*3/4…width][height*3/4…height]

B_0[0…width/4][0…height/4]  =  B[0…width/4][0…height/4]
B_1[0…width/4][0…height/4]  =  B[width/4…width*2/4][0…height/4]
   ⋮
B_15[0…width/4][0…height/4]  =  B[width*3/4…width][height*3/4…height]

sum_0:=0 ; sum_1:=0 ; sum_2:=0 ; …  sum_14=0; sum_15=0

for i:=0 to width/4  do
        for  j:=0 to height/4  do
                sum_0:=sum_0  +  A_0[i,j]  *  B_0[i,j]
                sum_1:= sum_1 +  A_1[I,j]  *  B_1[i,j]
                sum_2:=sum_2  +  A_2[i,j]  *  B_2[i,j]
                sum_3:=sum_3  +  A_3[I,j]  *  B_3[i,j]
                sum_4:=sum_4  +  A_4[i,j]  *  B_4[i,j]
                sum_5:=sum_5  +  A_5[I,j]  *  B_5[i,j]
                sum_6:=sum_6  +  A_6[i,j]  *  B_6[i,j]
                sum_7:=sum_7  +  A_7[I,j]  *  B_7[i,j]
                sum_8:=sum_8  +  A_8[i,j]  *  B_8[i,j]
                sum_9:=sum_9  +  A_9[I,j]  *  B_9[i,j]
                sum_10=sum_10  +  A_10[i,j]  *  B_10[i,j]
                sum_11:=sum_11  +  A_11[I,j]  *  B_11[i,j]
                sum_12:=sum_12  +  A_12[i,j]  *  B_12[i,j]
                sum_13:=sum_13  +  A_13[I,j]  *  B_13[i,j]
                sum_14:=sum_14  +  A_14[i,j]  *  B_14[i,j]
                sum_15:=sum_15  +  A_15[i,j]  *  B_15[i,j]

return sum_0+sum_1+….+sum_15
```

Figure 3.3: dot-product pseudocode after high-level transformation (width : width of A, height : height of A)

```
A[0…width]

haar_filter( A, N, L)
        w=N/2
        for i:=0 to w-1 do
                A'[i]=A[2i]+A[2i-1]
                A'[i+w]=A[2i]+A[2i-1]
        A=A'
        if L-1>0
                haar_filter( A, N/2, L-1)
```

```
A_0[0…width/16] = A[0…width/16]
A_1[0…width/16] = A[width/16…width*2/16]
        ⋮
A_15[0…width/16] = A[width*15/16…width]
w:=N/2

for i:=0 to w-1 do
    A_0[i]:= A_0[2i]  +  A_0[2i-1]
    A_0[i+w]:=A_1[2i]  +  A_1[2i-1]
    A_1[i]:= A_2[2i]  +  A_2[2i-1]
    A_1[i+w]:=A_3[2i]  +  A_3[2i-1]
    A_2[i]:= A_4[2i]  +  A_4[2i-1]
    A_2[i+w]:=A_5[2i]  +  A_5[2i-1]
          ⋮
    A_7[i]:= A_14[2i]  +  A_14[2i-1]
    A_7[i+w]:=A_14[2i]  +  A_14[2i-1]

    A_8[i]:= A_0[2i]  -  A_0[2i-1]
    A_8[i+w]:=A_1[2i]  +  A_1[2i-1]
    A_9[i]:= A_2[2i]  -  A_2[2i-1]
    A_9[i+w]:=A_3[2i]  +  A_3[2i-1]
    A_10[i]:= A_4[2i]  -  A_4[2i-1]
    A_10[i+w]:=A_5[2i]  +  A_5[2i-1]
          ⋮
    A_15[i]:= A_14[2i]  -  A_14[2i-1]
    A_15[i+w]:=A_15[2i]  +  A_15[2i-1]

for i:=0 to w-1 do
    A_0[i]:= A_0[2i]  +  A_0[2i-1]
    A_0[i+w]:=A_1[2i]  +  A_1[2i-1]
          ⋮
    A_3[i]:= A_6[2i]  +  A_6[2i-1]
    A_3[i+w]:=A_7[2i]  +  A_7[2i-1]

    A_4[i]:= A_0[2i]  -  A_0[2i-1]
    A_4[i+w]:=A_1[2i]  +  A_1[2i-1]
          ⋮
    A_7[i]:= A_6[2i]  -  A_6[2i-1]
    A_7[i+w]:=A_7[2i]  +  A_7[2i-1]

for i:=0 to w-1 do
    A_0[i]:= A_0[2i]  +  A_0[2i-1]
    A_0[i+w]:=A_1[2i]  +  A_1[2i-1]
    A_1[i]:= A_2[2i]  +  A_2[2i-1]
    A_1[i+w]:=A_3[2i]  +  A_3[2i-1]

    A_2[i]:= A_0[2i]  -  A_0[2i-1]
    A_2[i+w]:=A_1[2i]  +  A_1[2i-1]
    A_3[i]:= A_2[2i]  -  A_2[2i-1]
    A_3[i+w]:=A_3[2i]  +  A_3[2i-1]

for i:=0 to w-1 do
    A_0[i]:= A_0[2i]  +  A_0[2i-1]
    A_0[i+w]:=A_1[2i]  +  A_1[2i-1]

    A_1[i]:= A_0[2i]  -  A_0[2i-1]
    A_1[i+w]:=A_1[2i]  +  A_1[2i-1]
```

Figure 3.4: multi-layer haar pseudocode after high-level transformation (A : memory object input, N : length of the memory object, L : layer)

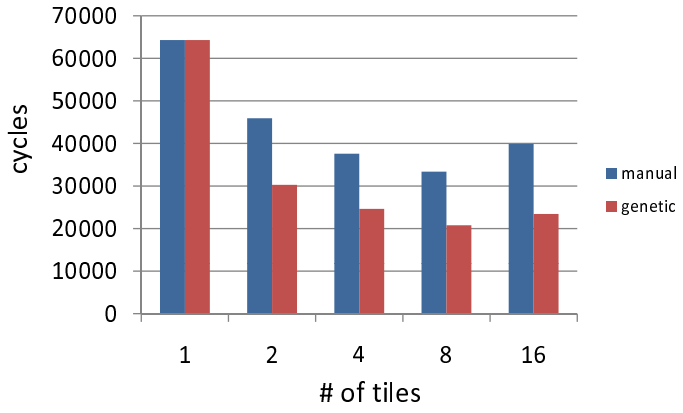


Figure 3.5: Execution time for convolution with varying numbers of tiles. The genetic algorithm runs for 100 generations with population size 200.

From the analysis of convolution assembly code, the manual placement results in serialized routing instructions, which cause low performance of object codes. According to the Raw profiler the version generated via manual placement spends 30,870 cycles while the version generated by the genetic algorithm spends only 21,107 on a part of serialized routing generation. Figure 3.7 illustrates how memory objects are arranged on 16 tiles. From this figure, we speculate that if the total size of memory objects assigned to a tile is significantly less than the data cache size, the load imbalance due to uneven distribution of memory objects is less important than the resulting impact on other instructions' and operands' placement, scheduling and routing.

In dot-product, $(sum = A[] \cdot B[])$, on two tiles and four tiles, the performance of the genetic algorithm is better than the manual placement by about 2,300 cycles and about 800 cycles respectively. This result is due to a relationship between a memory object C(16×16), which is only seen in assembly code and stores the result from dot-product of A(16×16) and B(16×16). According to instruction scheduling, a location of a scalar and a location of related memory object are not guaranteed to be in the same tile. On two tiles, locations of a series of half of scalars (`sum`) are different

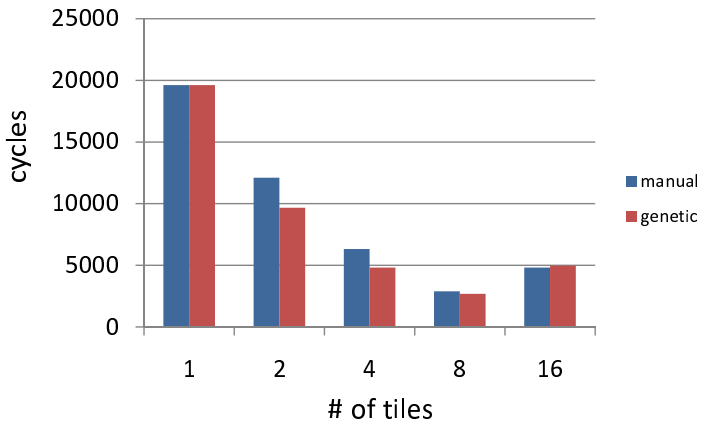


Figure 3.6: Execution time for dot-product. The genetic algorithm runs for 100 generations with population size 200.

from locations of the memory objects (c) when the compiler uses a manual placement. It generates serialized routing instructions at the end of a CFG node. Consequently, the routing instructions becomes a severe bottleneck, preventing other tiles from advancing to next CFG node. On the other hand, the genetic algorithm randomly assigns memory objects and avoids continuous serialized routing generation.

Contrary to previous results, an intuitive manual scheme on 8 tiles and 16 tiles performs better than the genetic algorithm. Dot-product shows the best performance when all memory objects related to computation belong to same tile (in this case, A and B in same tile). Even though the genetic algorithm prevents serialized routing instructions, it can not prevent the communication overhead between memory objects for dot-product computation. As more tiles are included in dot-product, memory objects are more dispersed over tiles in the genetic algorithm. We pay attention to a low probability of memory objects with the same tile locations because the genetic algorithm distributes memory objects at random. Tables 3.2 and 3.3 present how far memory objects are assigned among memory objects. Note that on 2 tiles, 13 sets of A and B are in same tile while only 2 sets of A and B are in same tile on

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	1	1	1	1	0	0	0	1	1	1	0	0	1	1	1	1
b	0	0	1	1	0	0	0	1	1	1	0	0	1	0	1	1
c	0	0	0	1	0	0	0	1	0	1	0	0	1	0	0	0

Table 3.2: Tile locations of memory objects on 2 tiles in a genetic algorithm of convolution

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
a	7	7	6	3	9	14	1	1	8	1	12	7	12	14	1	13
b	8	4	15	13	9	11	14	5	15	1	10	5	13	7	6	2
c	8	11	15	13	5	11	2	14	11	11	3	0	11	11	6	9

Table 3.3: Tile locations of memory objects on 16 tiles in a genetic algorithm of convolution

16 tiles. ON 16 tiles, Routing overhead between memory objects causes a decline in performance to compute $A[] \cdot B[]$.

After transformation, Haar includes only one matrix (16 distributed objects, `A0`..`A15`). The result indicates that the genetic algorithm is better than manual placement on 2 tiles, 4 tiles and 8 tiles. In the first phase of haar, all input objects are used. However, in the second phase of haar only half of the memory objects (`A0`..`A7`) are used in one CFG node while the rest of memory objects (`A8`..`A15`) are used in another CFG node. Haar continues until it has no memory objects as input. Manual placement keeps half of the tiles idle in the CFG node in first haar. Manual placement keeps only quarter of the tiles active in the CFG node in second haar. In the end, a tile with A0 is overloaded because it is utilized in every iteration of haar. From the control flow strategy of this compiler, we note that in a given CFG node, tiles with little scheduled work must wait for tiles that have more assigned work. This makes the genetic algorithm more suitable for haar because idle memory objects and active memory objects can

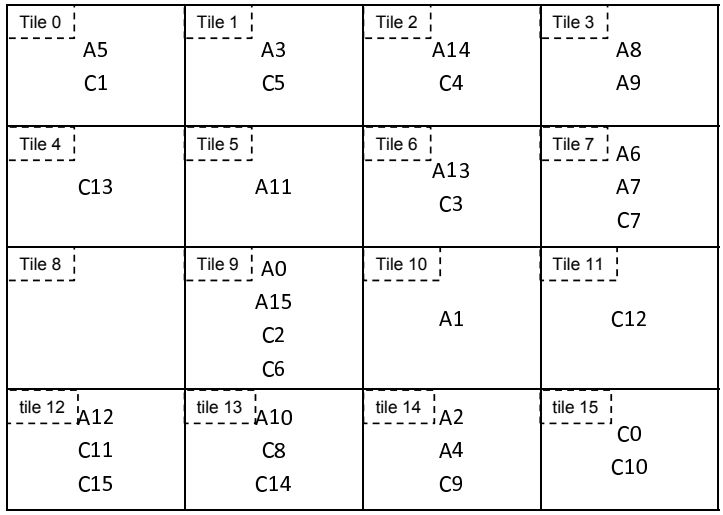
	idle (cycle)	active (cycle)
2 tiles	128	4224
4 tiles	128	3264
8 tiles	128	2488
16 tiles	128	1997

Table 3.4: Total execution cycles of an active tile and an idle tile in a CFG node of haar

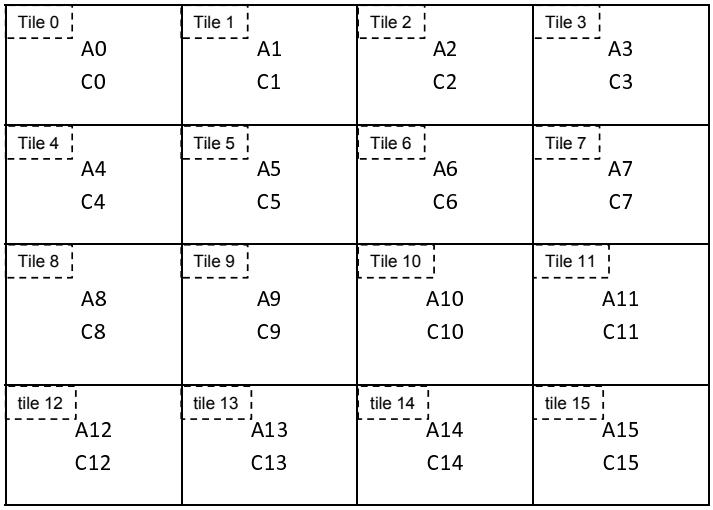
be placed in same tile within the CFG node. On 16 tiles, manual placement produces better qualified codes, because the impact of the imbalance is diminished as the work becomes more spread out. Thus, as memory objects are distributed, the idle time is also reduced. Table 3.4 shows that the execution time gap between idle tiles and active tiles keeps shrinking as the number of tiles increases in haar.

3.C.1 Performance improvement in more generations

Figure 3.9, 3.10 and 3.11 indicate that as generations elapse, the genetic algorithm produces faster executable codes. To help us analyze the performance improvement, three tables shows that overall speedups depend most on the reduction of receive-stalls. We observe how many gains are obtained in each case. After 100 generations on 16 tiles, receive-stalls change from 4,261 cycles from 3,496 cycles in dot-product. They decrease from 21,667 cycles to 16,934 cycles in convolution and from 3,081 cycles to 2,875 cycles in haar. As a result, the genetic algorithm makes executable codes that observe relatively less communication overhead between memory objects.



Genetic



Manual

Figure 3.7: Comparing the genetic algorithm and manual placement of memory objects on 16 files for convolution

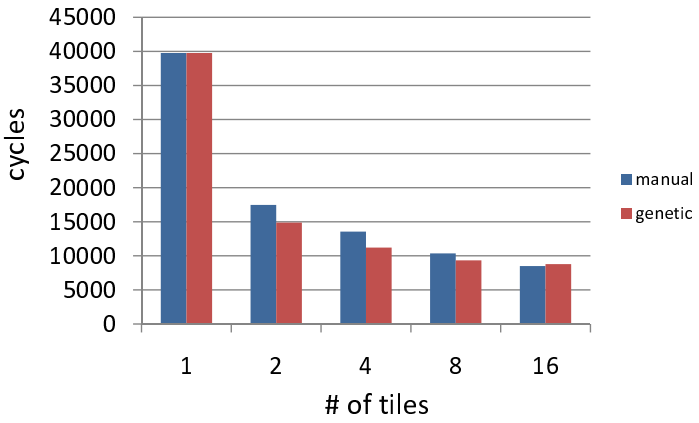


Figure 3.8: Execution time for haar. The genetic algorithm runs for 100 generations with population size 200.

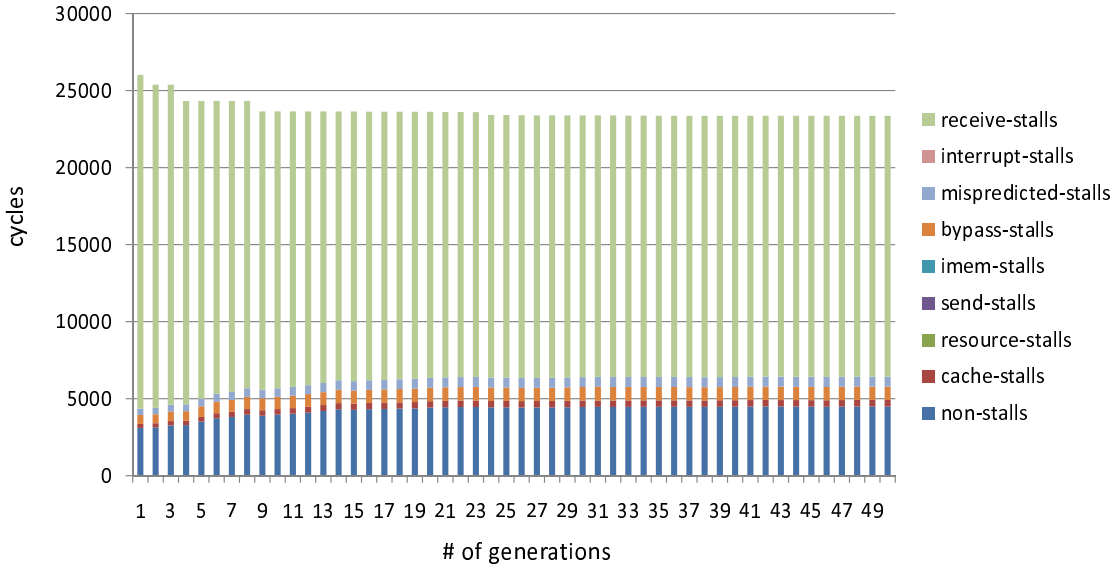


Figure 3.9: 50 generations of convolution on 16 tiles. The graph shows, for each generation, where cycles are spent in the most fit specimen in the population. Non-stalls are cycles spent successfully executing instructions. Cache-stalls are cycles spent cache missing. Resource-stalls (which do not occur in these programs) are cycles spent waiting for a functional unit to become available. Bypass-stalls are cycles spent waiting for a value to emerge from a local functional unit. Mispredicted-stalls are stalls caused by branch mispredictions. Interrupt-stalls (which do not occur here) are cycles lost because of interrupts. Send-stalls are cycles spent waiting for a network output port to have free buffer space. Finally, receive-stalls are cycles spent waiting for an incoming value. Interestingly, of these, send-stalls and receive-stalls are the stalls most optimized by changing memory object placement. In contrast, cache-stalls are relatively infrequent and thus do not constitute a significant enough factor in overall execution time.

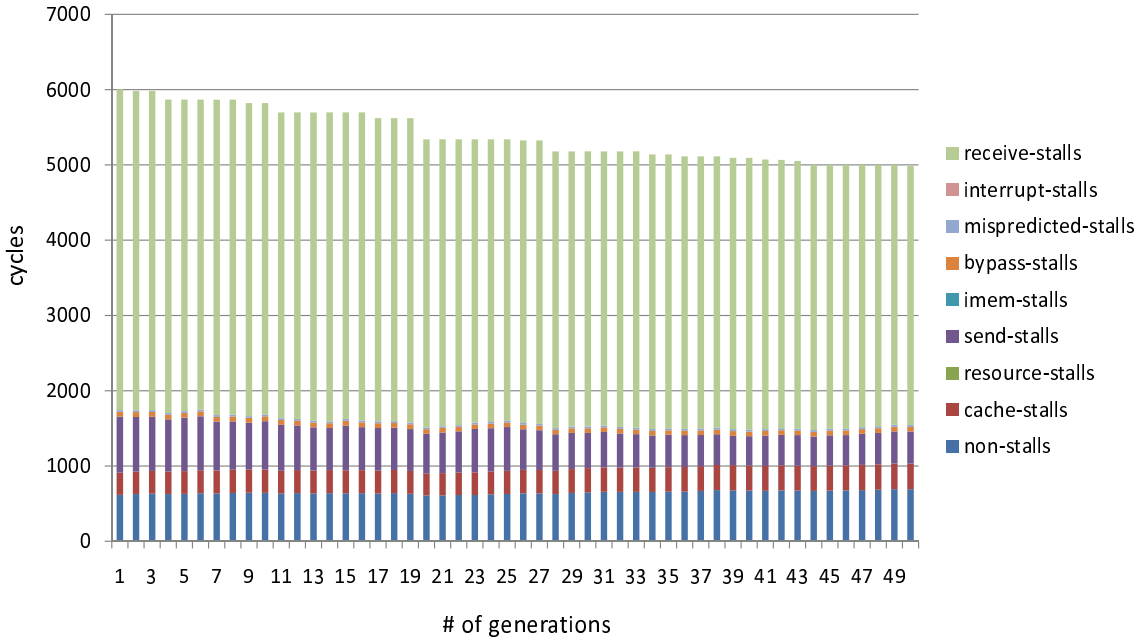


Figure 3.10: 50 generations of dot-product on 16 tiles

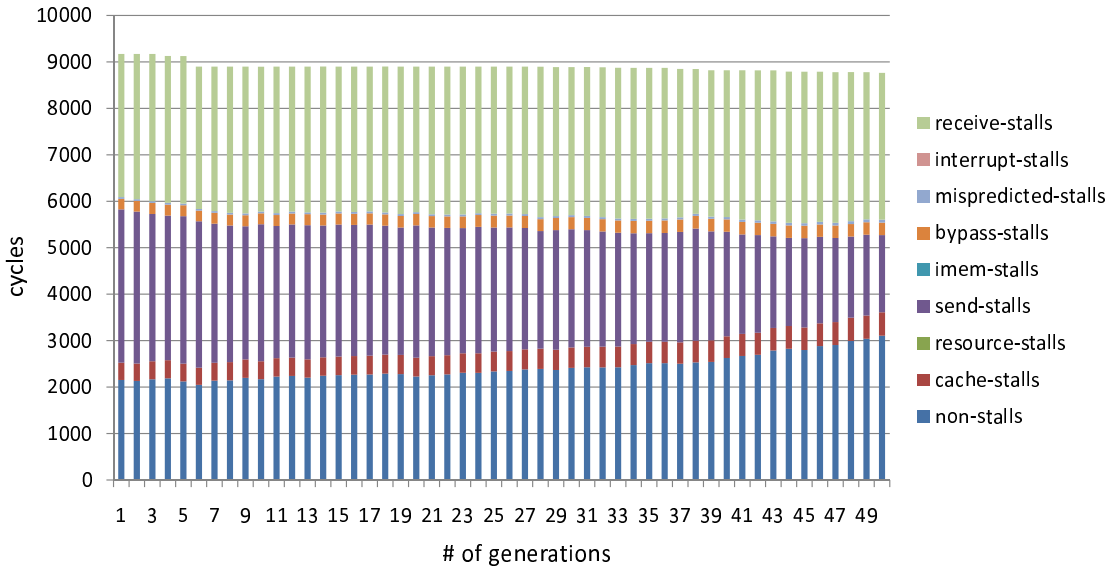


Figure 3.11: 50 generations of `haar` on 16 tiles

4

Conclusion

In the field of computer architecture, the trend has become to integrate exponentially more tiles into a single chip. Although single tile performance is largely dictated by microarchitecture, fast multi-tile performance can only be achieved through the use of a robust compiler and runtime infrastructure. In this thesis, we have proposed a methodology for constructing this infrastructure.

This thesis presents a complete compiler backend that generates parallel code for tiled microprocessors. It addresses complexity issues by separating the concerns of correctness and optimization. The optimization component uses standard machine learning algorithms (genetic programming), while the correctness component ensures that valid code is generated regardless of the input from the machine learning algorithm. The evaluation measures the compiler's ability to tune the placement of memory objects across tiles; in several cases it is able to perform placement better than a graduate student. Furthermore, it does this with no understanding, beyond what is necessary to generate correct code, of the particular target architecture (Raw).

Our compiler for tiled architectures includes several phases. In the *XML parser* phase, the compiler inputs two files and constructs the CFG graph. *IR translation* deals with instructions which depend on the hardware characteristics in the tiled architecture. In *home assignment*, all of memory objects are assigned to specific home

tile. In *scalar data analysis*, data dependency lists are created. In *instruction assignment*, all instructions have tile locations based on modified UAS. In *scalar location assignment*, tile locations are allotted scalars in instructions, live-in lists and live-out lists of a CFG node. In *stitch node insertion*, the compiler adds stitch nodes to force two adjacent CFG nodes to become consistent in live-in lists and live-out lists. In *routing instruction generation*, the compiler creates routing instructions, transporting scalars between tiles. In *register allocation*, a graph-coloring of the compiler replaces virtual registers with real ones. Finally, the compiler generates codes executable on every tile.

To evaluate the genetic algorithm for memory placement, we first evaluated a manual placement in which memory objects are evenly distributed across tiles. Then, we evaluated the use of a genetic algorithm in three benchmarks (convolution, dot product and haar). To obtain fitness values, we used execution time on a cycle-accurate simulator as the fitness function in order to attain more precise time measurements than heuristic in-compiler time approximations. As the generations elapse, the genetic algorithm converges with execution times that are close to the execution time of manual placement or better. Results of three benchmarks shows better performance in most cases. We observe that as the generation progress, performance is improved on most of tiles. In dot, a genetic algorithm shows improved performance on 2 tiles (19%) and 4 tiles (13%). In convolution, a genetic algorithm always shows better performance (35%–41%) than manual placement. On 16 tiles, it outperforms manual placement by 41%. In haar, the genetic algorithm outperforms manual placement by 15% on 2 tiles and 16% on 4 tiles.

For future work, the idea of using machine learning in the compiler to remove complexity could be further explored in the context of other NP-hard problems in tiled architecture compilation such as scalar assignment, instruction scheduling, routing generation and register allocation. Also, comparisons to existing algorithms to determine the net benefit in terms of complexity and quality of results could be evalu-

ated. A last interesting topic is to understand the impact of using compile-time fitness evaluation functions. This could potentially reduce the time required to evaluate a candidate program in the genetic algorithm.

Appendix A

Flow of instruction assignment

Figure A.1 depicts how to assign instructions across all tiles in compile-time. We sort all instructions by ready-time (ready-time refers to the time when an instruction is capable of occupying a computation unit). First, a list of data-ready instructions is formed.

Second, if the compiler acknowledges that a ready instruction has no data dependency with any other instructions, the compiler places the instruction in best slot among empty slots. A best slot refers to the location a place where an instruction can stay as close to center of tile configuration as possible. It avoids the worst distance from every other dependent tiles. It results in the acquisition of convincing average network latency from any tile. In Figure A.2, a ready-instruction will be placed into tile 5 in cycle time 1.

Third, if only one scalar in an instruction is dependent on another instruction, the compiler attempts to find the earliest empty slot in the same space column as an instruction with the scalar. It places a dependent instruction in the empty slot.

Fourth, if both scalars in an instruction rely on scalars in other instructions, the following different approach is adopted.

1. The compiler compares the ready time of two instructions producing scalar values and selects an instruction having a late ready time. It inserts a dependent

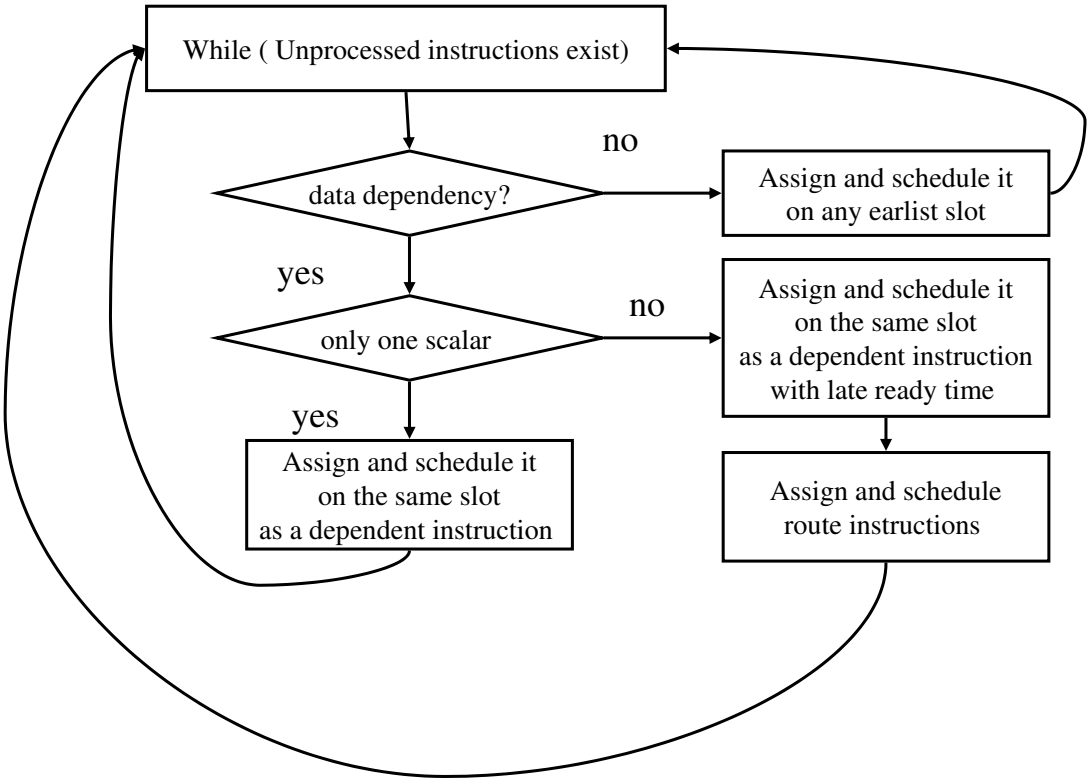


Figure A.1: A framework for instruction assignment

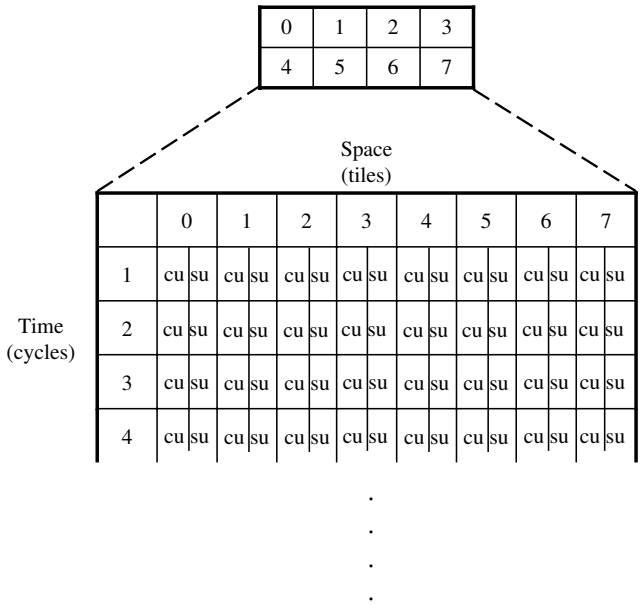


Figure A.2: A space-time map of 8 tiles (cu : a slot for a computation unit, su : a slot for a switching unit)

instruction in the same tile where the instruction was executed. The gap between routing time and time waiting for a late ready instruction might be compromised.

2. After routing instructions are generated in a map on dimensional order routing algorithm [6], they are inserted into slots of a map.

We have modeled a switch which holds only one switch instruction, without incoming buffers and outgoing buffers. If one switching instruction enters a slot in advance, any other instructions which attempt to enter the slot will be pushed into the next-cycle slot. It results in switching units changing idle status. Nevertheless, the current compiler supports this feature because it promises correct execution and prevents a deadlock in interconnection networks.

Appendix B

Stitch node insertion

In a CFG graph, two adjacent basic nodes, which are in the same control path, always have to remain consistent with a live-in list of a successor node and a live-out list of a predecessor node to ensure generation of correct routing instructions. However, in some cases, this premise might be broken. Let's take a look at Figure B.1.

In a loop, a live-in list of CFG node 2 should be equivalent to a live-out list of CFG node 1 with respect to scalar locations and virtual registers. Also, a live-out list of CFG node 3 should be identical to live-in lists of CFG node 2. (a) of Figure B.1 illustrates that the two conditions are contracted. To address this problem, a stitch node is inserted between CFG node 2 and CFG node 3 ((b) of Figure B.1). This stitch node contains routing instructions which transport scalar values between CFG nodes, that have different ideal scalar locations. A live-in list of a stitch node is copied from a live-out list of a predecessor of a stitch node. Conversely, a live-out list of a stitch node is obtained from a live-in list of a successor of a stitch node. From the live-in list and the live-out list, the stitch node generates routing instructions. As a result, consistency between CFG nodes is ensured through *stitch* nodes.

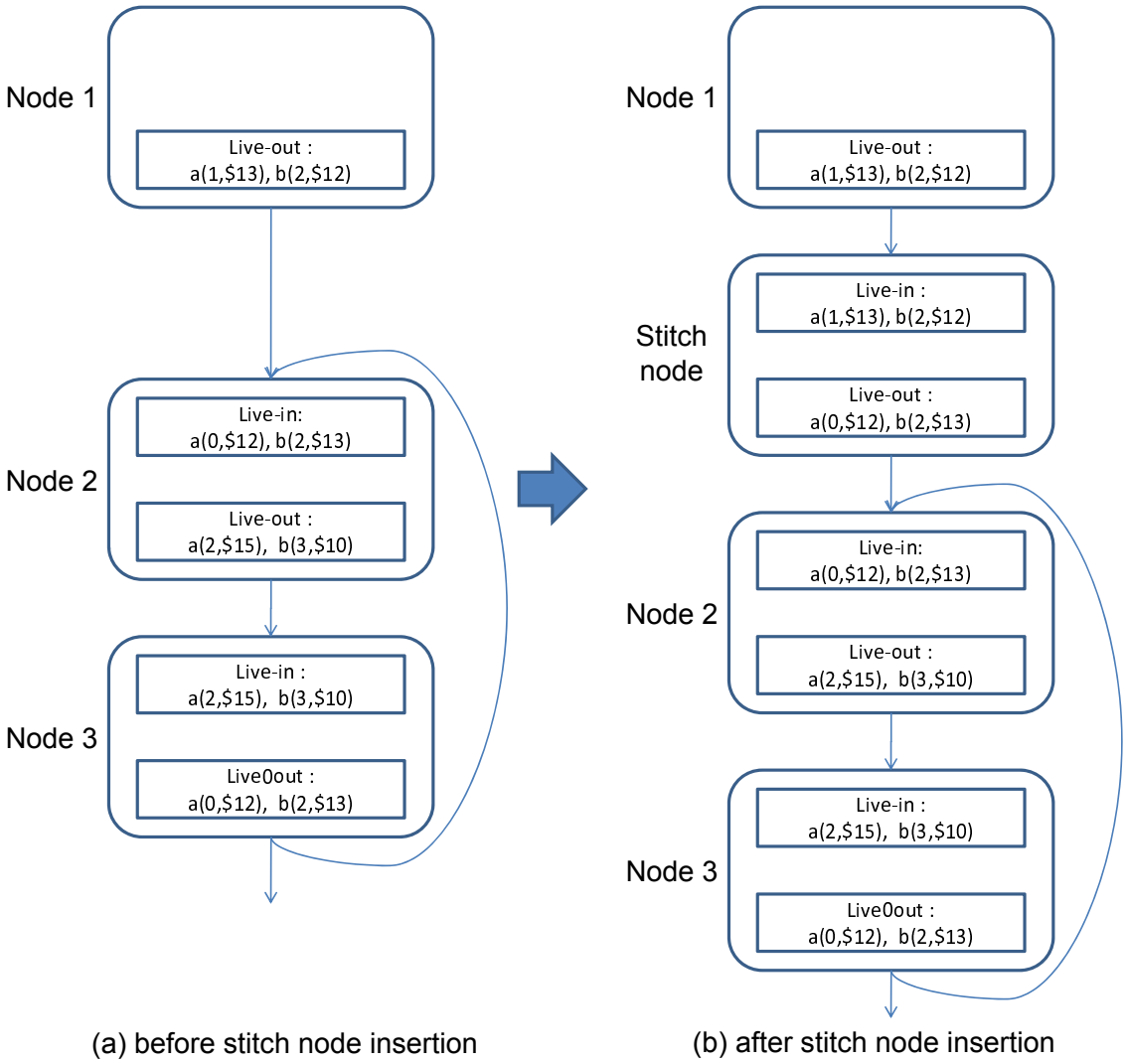


Figure B.1: A stitch node insertion example (a, b : scalars in a live-in list and a live-out list)

Appendix C

Routing instruction generation

Figure C.1 portrays how routing instructions are generated. Suppose that a scalar `a` of `mul` on tile 3 is dependent on scalar `a` of codeadd on tile 0. `mul` is aware of the location from which `a` has been retrieved, based on previous scalar assignment. Scalar `a` is transferred to a switch. On behalf of the computation unit, the switch on tile 0 sends the value to another switch via an interconnection network. In the end, a switch on tile 3 receives the value and delivers it to a computation unit.

Routing generation occurs in four regions within a CFG node. All routing generation must be carried out through one of four processes.

1. Some scalar values are transported from live-in lists to consuming instructions.
2. Some scalar values travel from producing instructions to consuming instructions.
3. Some scalar values are transferred from producing instructions to live-out lists.
4. The remaining of scalar values have to be sent from live-in lists to live-out lists.

All routing instructions created in this phase reside only within a CFG node. Inter-CFG routing instructions are eliminated through a consistent live-in list and live-out list of scalar assignment and *stitch* nodes.

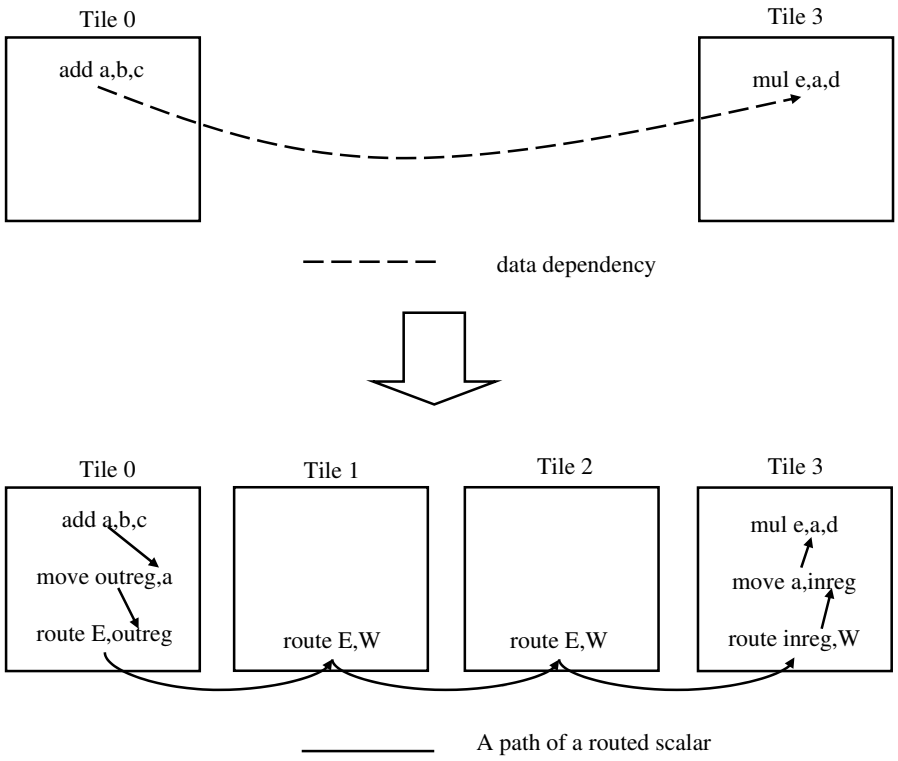


Figure C.1: An example of routing generation (`inreg` : an incoming register of a switching unit, `outreg` : an outgoing register of a switching unit, `W`(west), `E`(east) : routing directions, (`a` on tile 0, `e`) : def scalars, (`a` on tile 1, `b`, `c`, `d`) : use scalars)

Appendix D

Handling object migration in control flow

We have examined phases of the compiler in Chapter 2, and the issue of memory object placement in Chapter 3. In previous compilers, the “home” location of a memory object was fixed throughout the lifetime of the program. In the compiler described in this thesis, we allow memory objects to migrate between function calls, which can allow for greater scalability in the parallelization of large applications. If memory object migration is allowed, the issues of cache coherency and memory dependencies can arise. This section briefly overviews the solution that we employ in the compiler’s runtime in order to address this issue. The cost of this migration is accounted for in the genetic algorithm. However, due to time limitations, we were not able to thoroughly analyze the results, and so we describe the basic approach in this appendix.

D.A Allowing memory object migration

The compiler has assigned every memory object to ever-lasting tile locations in memory placement phase. The program with a CFG works thoroughly with assigned memory objects. If a CFG has functions inside, function calls in a CFG node may

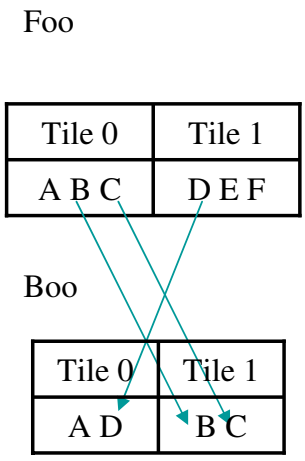


Figure D.1: Inconsistent memory object description

trigger a migration problem with respect to memory objects.

D.A.1 A problem

Suppose that function `Foo` (caller) calls `Boo` (callee)(It could be libraries outside or a function in `Foo`) and some memory objects are passed as arguments. Tile locations of memory objects in a caller might not be identical to ones of arguments in a callee. They might have different descriptions about memory locations, contradicting consistent memory locations in memory placement. In cache-incoherent system, which this compiler assumes, incorrect data in memory might be in use across function calls.

In Figure D.1, locations of `B`, `C` and `D` on `Foo` should be the same as `Boo` across function calls due to interprocedual analysis of memory placement. However, function calls put this principle in chaos. Through several analysis, the compiler assigns memory objects in a callee to tiles, which is different from tiles of a caller for better performance.

D.A.2 Remap function

To tackle this inconsistency, we propose a function called *remap*. It functions to make memory objects remain in consistent status at all time across function calls. The process is called *rehoming*. We have defined 3 data structures to implement remapping. These are only interfaces to help compiler writers to implement *remap* codes.

We also created new data structures to *rehome*. They facilitate the process of *ehome*. They are called Memory Object Description (MOD), Runtime Memory Object Description (RMOD) and architecture features (AF).

MOD includes elements below which are used mainly for analysis in compile-time.

- **Memory object name** : a scalar name in “def” IR or “inargs” IR
- **Tile number of Memory object** : a statically assigned tile location of a memory object

RMOD is defined for analysis in run-time. Elements are as follows.

- **Memory object size** : a size of a memory object which should be consistent across functions
- **Starting address of memory object** : an address where “Rehome” begins.

Last, AF defines attributes of tiled architecture.

- **Instruction latency** : instruction latency
- **Architecture cache type** : cache-coherent type or cache-incoherent type
- **SON Type** : a type of SON on tiled architecture

D.B How to handle it

In this section, we suggest some ways to deal with this problem. We limit ourselves in implementing “Remap” only if callees are inside a CFG. In library calls outside, we do not know the memory locations of callees. Therefore, if a caller has memory objects as arguments, all of memory objects are flushed into memory without intervention of the compiler. As for static strategies, there are three manners to deal with these problems - Migration in Compile-Time, Migration in Run-Time and Migration on SON. There are pros and cons in each way.

First, *Migration in Compile-Time* is to make them consistent on cache level in compile-time. This is convenient and fast in compiler implementation. There is no need to add management codes in a source. Instead, this scheme wastes time in flushing cache lines with same address repeatedly. Second, *Migration in Run-Time* is to make them consistent on cache level in run-time. This needs more implementation of compiler writer, even though execution-time may be faster than former. Last, *Migration on SON* leverages SON to migrate memory objects in register level via interconnection network.

- **Migration in compile-time**

  This handles memory object migration in compile-time. In Figure D.2, all of memory objects as arguments in a callee are flushed into memory, regardless of residing on cache. This benefits simple implementation for compiler writers. However, repeating redundant cache flushing is irresistible. We have adopted this scheme in current compiler.

  In Figure D.3, all of memory objects on data cache are flushed into memory. If a size of cache is much smaller than memory objects in tiles, this might show better performance

- **Migration in run-time**

  This addresses memory object migration in rum-time. It needs more complicated

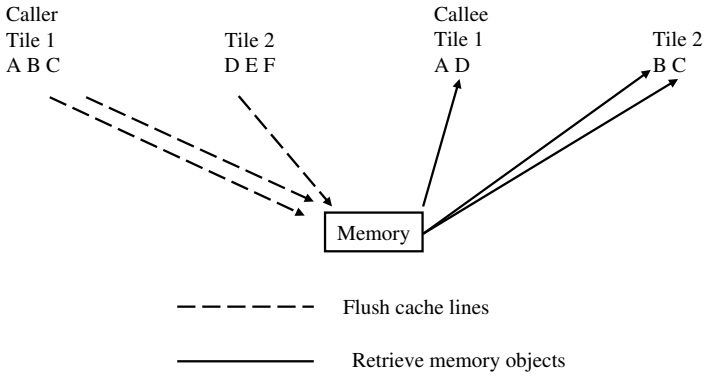


Figure D.2: The first migration in compile-time

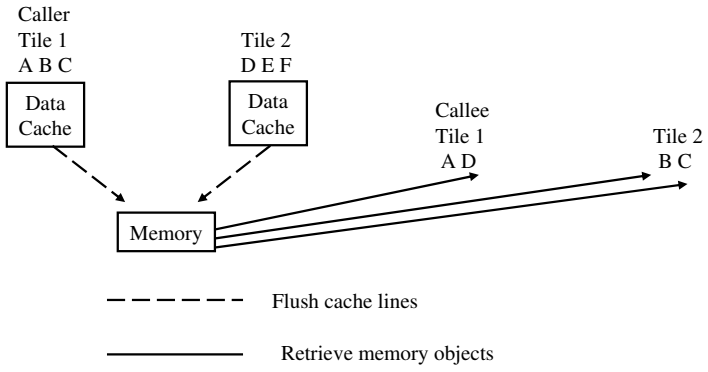


Figure D.3: The Second migration in compile-time

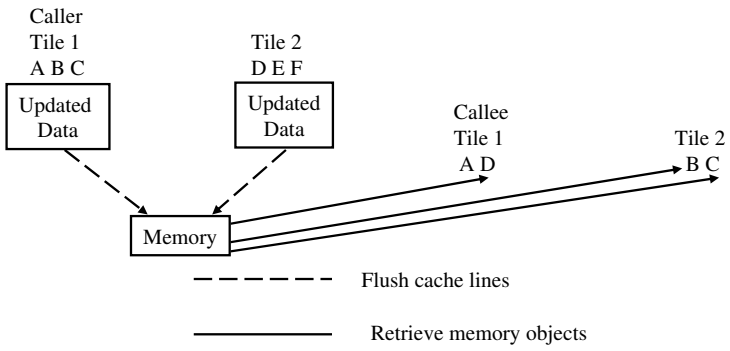


Figure D.4: The first migration in run-time

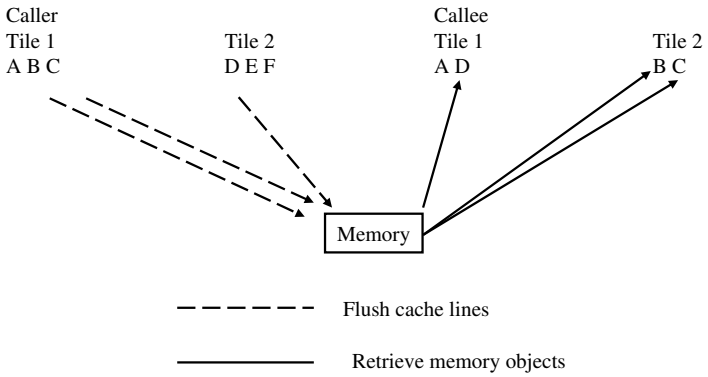


Figure D.5: The Second migration in run-time

analysis and attaches additional assembly codes for real-time analysis. Yet, the size of memory objects to be flushed is much smaller than methods in Figure D.2 and Figure D.3.

In Figure D.4, only cache lines with address, which must reflect changes in memory, are flushed into memory. Cache lines are not flushed into memory if memory objects with the address is only read by instructions or not accessed by other instructions.

In Figure D.5, only addresses with dirty data are flushed into memory. Control registers in tiled architecture enables the compiler to acknowledge if data in a

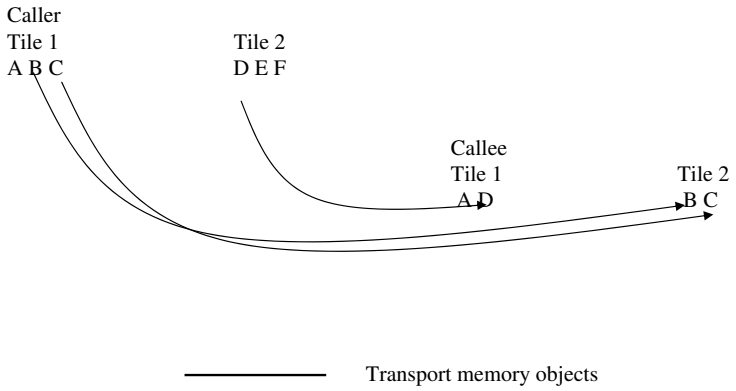


Figure D.6: A migration on SON

cache line is dirty or not. It totally depends on hardware characteristics of tiled architecture.

- **Migration on SON**

  The compiler runs migration via SON. This only manipulates register values on a chip, not accessing memory. If a size of memory objects to be flushed is quite small, this would be faster than flushing cache lines of memory objects one by one.

Bibliography

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1986.

[2] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Maps: A Compiler-Managed Memory System for Raw Machines. *ISCA '00: Proceedings of the International Symposium on Computer Architecture*, 2000.

[3] R. Barua, W. Lee, S. Amarasinghe, and A. Agarwal. Compiler Support for Scalable and Efficient Memory Systems. *IEEE Transactions on Computing*, 50(11):1234–1247, 2001.

[4] D. Burger, S. W. Keckler, K. S. McKinley, M. Dahlin, L. K. John, C. Lin, C. R. Moore, J. Burrill, R. G. McDonald, W. Yoder, and the TRIPS Team. Scaling to the end of silicon with edge architectures. *Computer*, 37(7):44–55, 2004.

[5] J. Cavazos, J. Eliot, and B. Moss. Inducing heuristics to decide whether to schedule. *SIGPLAN Not.*, 39(6):183–194, 2004.

[6] W. J. Dally and C. L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computing*, 36(5):547–553, 1987.

[7] L. George and A. W. Appel. Iterated register coalescing. In *POPL '96: Proceedings of the Symposium on Principles of Programming Languages*, pages 208–218, 1996.

[8] J. H. Holland. *Adaptation in natural and artificial systems*. MIT Press, Cambridge, MA, USA, 1992.

[9] F. Kri and M. Feeley. Genetic instruction scheduling and register allocation. In *QEST '04: Proceedings of the The Quantitative Evaluation of Systems, First International Conference on (QEST'04)*, pages 76–83, Washington, DC, USA, 2004. IEEE Computer Society.

[10] W. Lee, R. Barua, M. Frank, D. Srikrishna, J. Babb, V. Sarkar, and S. Amarasinghe. Space-time scheduling of instruction-level parallelism on a raw machine. *Architectural Support for Programming Languages and Operating Systems*, 32(5):46–57, 1998.

[11] R. Leupers. Instruction scheduling for clustered vliw dsps. In *PACT '00: Proceedings of the 2000 International Conference on Parallel Architectures and Compilation Techniques*, page 291, Washington, DC, USA, 2000. IEEE Computer Society.

[12] M. Mercaldi, S. Swanson, A. Petersen, A. Putnam, A. Schwerin, M. Oskin, and S. J. Eggers. Instruction scheduling for a tiled dataflow architecture. *SIGARCH Comput. Archit. News*, 34(5):141–150, 2006.

[13] G. E. Moore. Cramming more components onto integrated circuits. pages 56–59, 2000.

[14] E. Özer, S. Banerjia, and T. M. Conte. Unified assign and schedule: a new approach to scheduling for clustered register file microarchitectures. In *MICRO '98: Proceedings of the International Symposium on Microarchitecture*, pages 308–315, 1998.

[15] D. Puppin. Adapting convergent scheduling using machine learning. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, page 1, New York, NY, USA, 2003. ACM Press.

[16] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *MICRO '03: Proceedings of the International Symposium on Microarchitecture*, 2003.

[17] M. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The Raw Microprocessor: A Computation Fabric for Software Circuits and General-Purpose Programs. In *IEEE Micro*, March-April 2002.

[18] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar Operand Networks: On-Chip Interconnect for ILP in Partitioned Architectures. In *HPCA '03: Proceedings of the International Symposium on High-Performance Computer Architecture*, 2003.

[19] M. B. Taylor, W. Lee, S. Amarasinghe, and A. Agarwal. Scalar Operand Networks. In *IEEE Transactions on Parallel and Distributed Systems*, pages 145–162, February 2005.

[20] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring It All to Software: Raw Machines. *IEEE Computer*, pages 86–93, September 1997.