Analysis and Optimization of Paradigm Microprograms

Victor L. Winter¹, James McCoy², Dominic Montoya³, and Greg Wickstrom⁴

¹Department of Computer Science, University of Nebraska at Omaha, Omaha, NE, USA
²Sandia National Laboratories, Albuquerque, NM, USA
³Sandia National Laboratories, Albuquerque, NM, USA
⁴Sandia National Laboratories, Albuquerque, NM, USA

Abstract—Microcode often plays a key role in modern processor architectures. Microcode optimization is an important topic, and opportunities for microcode optimization can present themselves at various levels of abstraction. The Paradigm System, developed as part of a joint research effort between Sandia National Laboratories and the University of Nebraska at Omaha, consists of a high-level architecture-independent microprogramming language together with it’s compiler. This paper discusses the artifacts and mechanisms, within the Paradigm System, that support the analysis and optimization of Paradigm microprograms.

Keywords: micro-programming, microprogram optimization, microprogram analysis, program transformation

1. Introduction

On modern processing platforms there oftentimes exists a computational gap between the functionality provided by the assembly language instruction set, which is targeted by high-level language compilers, and the set of signals used to control hardware resources. In such an environment, microcode (μcode) can be effectively used to emulate the functionality of assembly language instructions that are not directly supported by the hardware.

Because μcode lies at the core of a processor’s design, it’s optimization is an important topic. Efficiency gains in μcode, even small gains, can have a substantive impact on system-level performance. Research into the optimization of μcode spans algorithmic optimization of high-level μcode down to determining the optimal order in which microoperations (μoperations) should be executed.

1.1 Application

Paradigm is a high-level architecture-independent microprogramming (μprogramming) language that has been developed as part of a joint research effort, funded by Sandia National Laboratories (SNL), between SNL and the University of Nebraska at Omaha. The primary application motivating Paradigm is the development of a processor, called the Scalable Core (SCore) [1]. The SCore is a hardware implementation of a subset of the JVM, designed and developed at SNL, for use in high-consequence embedded systems [2].

Within the SCore, the functionality of Java bytecodes is achieved through μprogramming. In particular, each Java bytecode supported by the SCore is realized through a corresponding μcode implementation. Furthermore, native methods used in the JVM and supported by the SCore are also implemented in μcode.

1.2 Contribution

The Paradigm System provides a unique environment for exploring μcode optimization through a mixture of manual activities such as restructuring high-level μprograms and automated activities such as the compaction of low-level μcode performed by the Paradigm compiler.

This article will discuss the following aspects of the Paradigm System that facilitate μcode analysis and optimization.

- extensive analysis – The Paradigm compiler produces a variety of artifacts (heat maps, graphs, tables) providing developers with insight into the allocation of registers that occur during compilation as well as detailed estimates of time/space trade-offs associated with calling versus in-lining methods.
- user defined optimizations – Paradigm developers can affect compilation in three important ways: (1) through in-lining directives, which are source-code level directives instructing the compiler to inline particular methods, (2) through optimizing transformations, which are transformation rules that developers can add to the compiler itself in order to perform specific optimizations during compilation, and (3) through timing constraints, which are used to guide the compaction of μcode instructions.

The remainder of this article is structured as follows: Section 2 reviews some of the basic concepts and terminology of microcoding. Section 3 overviews related work. Section 4 discusses analysis artifacts produced by the Paradigm compiler. Section 5 overviews user-defined optimization rules that can be added to the Paradigm compiler. Section 6 describes the declarative language used by Paradigm to specify parallel capabilities of a target machine, and Section 7 concludes.
2. The Basics of Micro-programming

The purpose of $\mu$programming is to orchestrate the behavior of resources in a CPU. The basic concept was developed by Maurice Wilkes [3] in 1951 who also coined the term micro-programming. Microprogramming ($\mu$programming) provides what amounts to a software-based alternative to the hardware-based logic boxes whose design was (and is) considered to be a bit of a black art[4].

A $\mu$program is a specification of how the resources within a CPU are to be controlled. $\mu$programs can be expressed at various levels of abstraction. High-level $\mu$programs strive to facilitate human comprehension, and can have syntactic and semantic similarities to high-level general-purpose programming languages such as C and Java. In contrast, low-level $\mu$programs are suitable for execution on a processor. The purpose of a $\mu$compiler is to translate a high-level $\mu$program into a low-level $\mu$program.

A low-level $\mu$program consists of a sequence of $\mu$instructions. A $\mu$instruction consists of a set of $\mu$operations each of which specify the control of a fundamental resource within the processor. Typical examples of $\mu$operations include:

- the transfer of data from memory to a register
- elementary operations such as shift, load, and clear performed on data residing a register
- properly updating the internal registers of the control unit in order to enable a jump

A $\mu$operation consists of a set of fields. Fields are made up of bits whose binary values correspond to control lines. For example, a field consisting of $k$ bits can be used to denote $2^k$ combinations of control lines. A special case arises when $k = 1$ for all fields. $\mu$instructions constructed exclusively from $\mu$operations having 1-bit fields are referred to as horizontal $\mu$instructions. Horizontal $\mu$instructions are long, but allow for the maximal expression of parallelism. In contrast, the signals denoted by fields for which $k > 1$ are encoded, and $\mu$instructions made up of such fields are referred to as vertical $\mu$instructions. A benefit of such encoding is that the bit-width of $\mu$instructions is significantly reduced. However, the parallelism which can be expressed through vertical $\mu$instructions is limited and combinatorial logic is needed to decode field values.

Orthogonal to the vertical/horizontal nature of a $\mu$instruction is the architectural notion of how many $\mu$operations a $\mu$instruction can hold. If only a “few” $\mu$operations can be placed into a $\mu$instruction the machine has a vertical architecture; otherwise it has a horizontal architecture[5][6].

For architectures that support concurrent execution of $\mu$operations, be they vertical architectures or horizontal architectures, the scheduling of $\mu$operations presents an area of optimization. In this context, the goal of optimization is to produce a low-level $\mu$program having a minimal or near-minimal number or $\mu$instructions. For this form of optimization, referred to as $\mu$code compaction, achieving optimal results has been shown to be NP-complete[7]. There are two types of compaction: (1) local compaction which focuses on restructuring the $\mu$operations within straight-line $\mu$code(SLM) – also known as basic blocks, and (2) global compaction whose focus spans multiple SLMs.

3. Related Work

Research into the design of high-level $\mu$programming languages and $\mu$compilers predominantly took place during the 1970’s and early 1980’s. A number of papers have been published on the topic of $\mu$code optimization [8], [9], [10]. Aghwala [11] has written a survey on $\mu$code optimization. A central issue in the type of optimization discussed in the survey is the reduction of the size of the control memory needed to hold a $\mu$program implementing a given function. Here, the control memory is modeled as a two-dimensional array $(W \times B)$ where $W$ denotes the number of words (i.e., rows) and $B$ denotes the number of bits (i.e., columns) in the control memory respectively. A primary goal of optimization is to reduce the control memory along either of its dimensions.

In [11], optimization strategies are categorized as being either high-level or low-level. High-level optimizations are based on dataflow analysis of the source-code and strive to discover parallelism inherent in the algorithm implementation. Optimizations possible at this level also include existing (well-known) compiler optimization techniques. Roughly stated, the result of high-level optimization is a sequence of sets, called time frames, whose elements are $\mu$operations. This sequence of time frames is viewed as partitioning the computation defined by the high-level (input) $\mu$program in a manner that is maximally parallel irrespective of physical limitations of the host machine. After such a partitioning has been completed, low-level optimizations can be applied to map the structure onto a host machine. These low-level optimizations center on normalizing the existing partition structure so that each set in the partition can be realized by exactly one horizontal $\mu$instruction.

SIMPL (Single Identity Microprogramming Language) [12] is a high-level (machine dependent) $\mu$programming language developed in the early 70’s having an ALGOL-like syntax. During SIMPL compilation, a high-level sequential program undergoes sophisticated analysis in order to produce a highly optimized low-level horizontal program. SIMPL optimization is based heavily on the single identity principle which states that a (particular) definition for a variable holds from the point it is assigned up to the point where it is reassigned. The single identity principle forms the basis for partitioning a sequence of statements into subblocks each of which constitutes an independent set of $\mu$operations. This decomposition represents a key first step in solving the global optimization problem.
Though there were a number of µcode language and compiler development efforts underway at the time, SIMPL was considered to be the first high-level µprogramming language in which both compilation and optimization were performed automatically. A SIMPL compiler has been developed targeting the Tucker-Flynn dynamic microprocessor [13].

Micro-C [14] is a high-level machine-independent µprogramming language compatible with C. A Micro-C µprogram can be compiled by a special compiler based on the Portable C Compiler. The output produced by this compiler is vertical (i.e., unoptimized) symbolic µcode. This intermediate representation can then be optimized by a “straight-line” packer which translates sequences of µoperations into horizontal µinstructions. An assembler is then used to translate the result into executable low-level µcode.

In [15], a language is presented in which high-level µprograms are composed of declaration statements and command statements. The compiler for this language consists of two phases: In the first phase of compilation, the input µprogram is parsed, analyzed, and an unoptimized sequence of µinstructions is produced. At this stage, each µ-instruction performs exactly one elementary operation (i.e., a µoperation). The second phase of compilation is an optimization phase in which a number of tables containing machine-dependent information (e.g., parallel capabilities of the hardware) are employed in order to compact µinstructions taking full advantage of the parallel capabilities of the hardware.

In [16], an approach is presented where machine-independent high-level µcode optimization is performed by the software component of a µcode compiler and low-level machine-dependent optimization is performed by hardware residing on the host machine (i.e., the machine on which the µcode will be executed). In this context, the goal of a hardware microcode optimizer (HMO) is to condense a sequence of µinstructions (i.e., where each µinstruction contains only one µoperation) into a functionally equivalent sequence of µinstructions taking full advantage of the parallel capabilities of the host machine. At a higher-level, optimization strategies are divided into two distinct categories: The local optimization category is performed by the hardware-based component of the compiler and focuses on the serial combination (i.e., compaction) of µinstructions. The global optimization category is performed by the software-based component of the compiler and focuses on the commutative reordering µinstruction sequences (driven by dataflow analysis) in order to more fully exploit parallelism.

4. Analysis

In addition to designing well-structured high-level µprograms, developers often need to pay close attention to the consumption of resources entailed by their design. For example, how many internal registers are needed by the compiler to compile a given high-level µprogram? What is the size, in terms of the number of µinstructions, of the resulting low-level µprogram produced by the compiler? And, how many µinstructions are executed when the program is run?

The Paradigm compiler produces three artifacts to assist developers in their optimization-oriented analysis efforts: (1) views, (2) heat maps, and (3) estimation tables.

4.1 Views

Paradigm provides a notation, called a view, for specifying subsets of methods. From the specification of such subsets, views can be constructed. In particular, a view is an acyclic directed graph whose nodes denote methods and whose labeled edges denote the number of internal registers allocated by the compiler relative to specific nodes. For example, consider the graph below consisting of two nodes, labeled f and g, connected by an edge labeled 12.

```
[12]  f  −−−−−−−−→  g
```

This graph indicates that (1) the method g is called in the body of f, and (2) at the point of the call to g, the compiler has allocated 12 internal registers local to the context of f.

A high-level µprogram may have multiple views defined for it, each of which will be output to a correspondingly named file. Such files are output in a “dot format” and can be viewed using Graphviz. Figure 1 shows an example of a view generated by the Paradigm compiler for a µprogram produced for a hypothetical machine.

4.2 Heat Maps

Heat maps are another form of feedback produced by the Paradigm compiler. Specifically, the Paradigm compiler will output twelve attributes to a file in a comma-separated value format. Attributes range from method arity, method size, reference_frequency, inline - called size, to (inline - called size) * reference_frequency. Figure 2 shows a heat map for a hypothetical machine. In this heat map, the first grouping (in grey) is by method type (e.g., macro, subroutine, operator, operation, condition, interface). The second grouping (also in grey) is the difference between the in-lined size and the called size – this includes all overhead associated with making a method call. The size of squares in the heat map represents the called size, and the color indicates reference frequency with red denoting the most frequently referenced methods and blue denoting the least frequently referenced methods.

4.3 Efficiency Estimator

In order to meet resource constraints, it may be necessary for developers to optimize their high-level µprogram. To facilitate optimization, Paradigm provides high-level language directives that can be used to instruct the compiler to in-line various method declarations.
Method in-lining represents a time/space tradeoff, since in-lining can cause the size of the low-level \( \mu \text{code} \) to expand dramatically. For example, suppose the body of a method \( m \) consists of 100 lines of \( \mu \text{code} \). Further suppose that \( m \) is called in 10 places in the \( \mu \text{code} \). If all 10 calls are in-lined, then in-lining (without compression) will yield 1000 \( \mu \text{instructions} \). In contrast, suppose that a \( \text{call} \) to the method \( m \) requires 20 lines of \( \mu \text{code} \). In this case, calling \( m \) 10 times will result in a total overhead of 200 lines of \( \mu \text{code} \). Thus, an implementation in which \( m \) is called will contain 700 fewer lines of \( \mu \text{code} \)(i.e., 200 lines of call overhead plus 100 lines for the method body). However, it should be noted that in-lined methods always execute faster than their called counterparts since there is no call overhead associated with their execution.

The overhead associated with a method call is significant. Internal registers must be allocated for the input parameters. Instructions must be generated by the compiler to \( \text{move} \) actual parameters to the internal registers corresponding to formal input parameters of the method. A \( \text{call} \) instruction must be generated by the compiler to transfer execution to the method body, and a \( \text{return} \) must be executed upon completion of the method body. Furthermore, moves, calls, and returns do not lend themselves to compression. In other words, only one such \( \mu \text{operation} \) will fit into a \( \mu \text{instruction} \). Thus, going back to our previous example, if the execution of each \( \mu \text{instruction} \) takes 1 unit of time, then executing the body of \( m \) via a call will take 120 units of time.

As the body of a method gets smaller it gradually becomes more attractive to in-line a method. Eventually, a crossover point is reached where calling a method consumes more time and more space than simply in-lining a method. It should be noted that, from the point of view of development, the method is a mechanism for abstracting functionality. Thus, a best-practices approach to development would encourage the use of methods as needed to give clarity to an implementation.

The \textit{Paradigm} compiler, provides an estimation of the effects of method call versus method in-lining. In particular, two sorted tables are produced: (1) a \textit{static call-frequency estimation table}, and (2) an \textit{execution path estimation table}. Examples describing the information in both of these tables are described in the sections that follow.

### 4.3.1 Example: Static Call-Frequency Estimation

Suppose method \( m_1 \) is an in-line method candidate having 3 formal input parameters. Furthermore, let us assume that a static inspection of the \textit{Paradigm} application reveals that \( m_1 \) is called from 10 syntactically distinct locations. Similarly, suppose method \( m_2 \) is inline candidate method having 2 formal input parameters. Furthermore, let us assume that a static inspection of the \textit{Paradigm} application reveals that \( m_2 \) is called from 20 syntactically distinct locations.

<table>
<thead>
<tr>
<th>Static Call-Frequency Estimation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method</td>
</tr>
<tr>
<td>--------</td>
</tr>
<tr>
<td>( m_1 )</td>
</tr>
<tr>
<td>( m_2 )</td>
</tr>
</tbody>
</table>
It should be noted that static-call-estimation provides a fairly coarse grained and basic estimation of the overhead associated with calling methods. In particular, static call estimation does not take into account execution paths which can have multiplicative effect on the number of times a method can actually be called during runtime. For example, suppose method \( m1 \) is called twice in the body of method \( m2 \), and suppose method \( m2 \) is called 5 times within the \( \mu \)code. Note that in this example, there are only 2 lexical occurrences of \( m1 \). However, \( m1 \) will be called a total of \( 5 \times 2 = 10 \) times during the execution of the application. We call this second form of estimation execution path estimation. It should be noted that, since it does not account for loop iterations, execution path estimation is also only an estimate, albeit a more accurate one than static call estimation.

4.3.2 Example: Execution Path Estimation

Suppose methods \( m1, m2 \) and \( m3 \) are respectively called 5, 6, and 4 times from the \( \mu \)code as shown in Figure 3. Also note, that \( m1 \) is called 2 times from \( m2 \) and \( m2 \) is called 3 times from \( m3 \).

The execution path estimation table shows that the total calls for \( m1 \) is 41. This value corresponds to the sum: \( 1 \times 5 + 1 \times 6 + 2 + 1 \times 4 + 3 \times 2 = 41 \). More specifically, \( m1 \) is called 5 times from the \( \mu \)code. This accounts for the \( 1 \times 5 \) term. Next, \( m1 \) is called 2 times from \( m2 \), which itself is called 6 times from the \( \mu \)code. This accounts for the term \( 1 \times 6 \times 2 \). And finally, \( m2 \) is called 3 times from \( m3 \) which is called 4 times from the \( \mu \)code. This accounts for the term \( 1 \times 4 \times 3 \times 2 \).

In this example, the in-lined size for \( m1 \) is 0. This is because the body of \( m1 \) is empty (after the removal of the return instruction). The called size for \( m1 \) is 83 and corresponds to \( 41 \) calls plus \( 41 \) returns plus the size of the declaration of \( m1 \) (which is 1). The in-lined time will always be equal to the in-lined size. The assumption here is that each row in the \( \mu \)code takes 1 unit of time to execute, and that additional compression of method bodies is not possible.

The called time for \( m1 \) is 82. This corresponds to the total calls to \( m1 \) times the sum of the number of moves associated with calling \( m1 \) plus the number of microcode rows associated with the call-to and return-from \( m1 \).

And finally, the speed up is 100%. This number is computed using the following formula:

$$ 100.0 - \frac{(inlined\_execution\_time/called\_execution\_time) \times 100.00}{-} $$

Although it is not highlighted by the example given, it should be noted that the execution path estimator accounts for the mandatory inlining of all macros, interfaces and

<table>
<thead>
<tr>
<th>Method</th>
<th>Total Calls</th>
<th>Inlined</th>
<th>Called</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m1 )</td>
<td>41</td>
<td>0</td>
<td>83</td>
</tr>
<tr>
<td>( m2 )</td>
<td>18</td>
<td>36</td>
<td>39</td>
</tr>
<tr>
<td>( m3 )</td>
<td>4</td>
<td>12</td>
<td>12</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Method</th>
<th>Inlined</th>
<th>Called</th>
<th>% Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td>( m1 )</td>
<td>0</td>
<td>82</td>
<td>100%</td>
</tr>
<tr>
<td>( m2 )</td>
<td>36</td>
<td>72</td>
<td>50.0%</td>
</tr>
<tr>
<td>( m3 )</td>
<td>12</td>
<td>20</td>
<td>40.0%</td>
</tr>
</tbody>
</table>

Table 1: Execution path estimation.

```
interface call(LabelType toLabel) { aux_call(); }
interface return() { aux_return(); }

subroutine m1() returns void { return(); }
subroutine m2() returns void {
    m1(); m1(); return();
}
subroutine m3() returns void {
    m2(); m2(); return();
}

microcode {
    m1(); m1(); m1(); m1();
    m2(); m2(); m2(); m2(); m2(); m2();
    m3(); m3(); m3(); m3();
}
```

Fig. 3: Example used for execution path estimation

condition\(^1\). This is important because such mandatory inlining can result in dramatic changes in the final size of a subroutine or operator. Also note that the size of the call and return interfaces also take inlining into account.

5. User-defined Optimization Rules

The Paradigm compiler is transformation-based and implemented in the TL System\[17\]. During compilation, a Paradigm program is passed through a number of canonical forms, each of which can be output in human-readable form. The Paradigm compiler is extensible in the sense that it supports the incorporation of user-defined transformation rules into the compilation process. Such rules provide domain experts the opportunity to perform custom optimizations specific to a particular architecture or \( \mu \)code design. Figure 3 is an example of a \( \mu \)code fragment, which can be output by the compiler, consisting of a sequence of \( \mu \)operation method calls separated by labels denoting jump destinations (e.g., starting positions of methods whose bodies have not been in-lined).

By inspection of the sequence of operations we see that a writeReg operation is immediately followed by a copyReg operation. Suppose that by combining knowledge of the

\(^1\)The language Paradigm has five different kinds of methods. The rational behind this is beyond the scope of this article.
6. Paradigm’s Timing Constraint Language

Paradigm provides a declarative language, called TCL, for specifying the timing constraints of a targeted hardware architecture. Timing constraints form the basis of a local compaction algorithm focusing on the compression of straight-line µcode (SLM). Timing-constraint based optimization does not involve commutative reordering of µoperations, instead it focuses on maximizing the compression of adjacent (i.e., associative) µinstructions. It is worth mentioning that in the compilation stage where timing-constraint based optimization occurs, the µprogram being compiled is in a form where all non-sequential control flows are expressed in terms of jumps to labels. In this context, an SLM is then simply the sequence of µinstructions occurring between consecutive labels.

Conceptually, a timing constraint is a pair of logical formulas that, if satisfied by adjacent µinstructions, prevent them from being compressed into a single µinstruction. Compression is also (implicitly) prohibited in cases when corresponding fields, in adjacent µinstructions, contain distinct (i.e., unequal) non-default signals.

An abstract example of the syntax of a timing constraint is shown in Figure 6. In the example, $F_1$ and $F_2$ denote the pair of logical formulas of the timing constraint named $TC_k$.

The evaluation of $TC_k$ with respect to a pair of adjacent µinstructions $I_j$ and $I_{j+1}$ proceeds as follows: If $I_j$ satisfies $F_1$ and $I_{j+1}$ satisfies $F_2$, then we say that the the µinstructions $I_j$ and $I_{j+1}$ satisfy the timing constraint $TC_k$, in which case the compression of $I_j$ and $I_{j+1}$ is prohibited by $TC_k$; otherwise compression is not prohibited by $TC_k$.

Fig. 6: An abstract example of a timing constraint.

TCL allows µcode compression to be restricted by a set of timing constraints $S_{TC} = \{TC_1, \ldots, TC_m\}$. The compression of any pair of µinstructions $I_j$ and $I_{j+1}$ is prohibited if $\exists TC_k \in S_{TC}$ such that $TC_k$ is satisfied by the µinstructions $I_j$ and $I_{j+1}$.

A more detailed look at timing constraints reveals that they are logical formulas, in conjunctive normal form, whose elements are equality/inequality matching-based comparisons involving fields. An abstract example of a disjunction constraining the fields $f1$ and $f2$ is shown below.

field.f1 = field1Type.item1 | field.f2 != field2Type.item2

Within an element, there are three kinds of items that can be associated with a fieldtype: (1) a symbolic name denoting a constant value belonging to a type declaration, (2) a subscripted variable which can match with field constants (occurring in the µinstructions in which evaluation is taking place), and (3) the keyword DEFAULT/NONDEFAULT. The scope of a subscripted variable spans an entire constraint (both formulas) and can therefore be used to express equality-based properties between fields within a constraint.

The Paradigm compiler provides feedback summarizing the impact of the optimizations it performs. Figure 7 shows an example of an optimization summary.

7. Conclusion

In the design of a high-level architecture-independent µprogramming language, a major issue that must be confronted centers on how architecture-specific information can
be specified, as well as how the compiler for the language can utilize this information to produce efficient low-level microcode targeting a host machine. Addressing this issue, Paradigm provides a timing constraint language (TCL) for specifying the parallel capabilities of a host machine. Furthermore, the Paradigm compiler also provides extensive feedback on the nature of its compilation, including pretty-printed representations of the program being compiled during various stages of compilation, register usage, call frequency, and comparisons between overheads associated with method call versus method in-lining. This information can be used to guide time/space optimizations involving design level decisions such as method in-lining and can even guide the development of user-defined rule-based application-specific optimizations that can be folded into the compilation process itself.

References


