SAND2014-1774C

Analysis and Optimization of *Paradigm* Microprograms

Victor L. Winter
Department of Computer Science
University of Nebraska at Omaha
USA
Email: vwinter@unomaha.edu

James McCoy, Dominic Montoya, and Greg Wickstrom
Sandia National Laboratories
USA
Email: {damonto,glwicks}@sandia.gov

***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**

I. 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.

A. 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)* [10]. The SCore is a hardware implementation of a subset of the JVM, designed and developed at SNL, for use in high-consequence embedded systems [16].

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*.

B. 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 II reviews some the basic concepts and terminology of microcoding. Section III overviews related work. Section IV discusses analysis artifacts produced by the *Paradigm* compiler. Section V overviews user-defined optimization rules that can be added to the *Paradigm* compiler. Section VI describes the declarative language used by *Paradigm* to specify parallel capabilities of a target machine, and Section VII concludes.

II. The Basics of Micro-programming

The purpose of *μprogramming* is to orchestrate the behavior of resources in a CPU. The basic concept was developed by Maurice Wilkes [14] in 1951 who also coined the term *micro-programming*. Microprogramming (*μprogramming*) 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[6].

A *μprogram* is a specification of how the resources within a CPU are to be controlled. *μprograms* can be expressed at various levels of abstraction: *High-level μprograms* 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 μprograms* are suitable for execution on a processor. The purpose of a *μcompiler* is to translate a high-level *μprogram* into a low-level *μprogram*.

A low-level *μprogram* consists of a sequence of *μinstructions*. A *μinstruction* consists of a set of *μoperations* each of which specify the control of a fundamental resource within the processor. Typical examples of *μoperations* 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 *μoperation* 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. *μinstructions* constructed exclusively from *μoperations* having 1-bit fields are referred to as *horizontal μinstructions*. Horizontal *μinstructions* are long, but allow for the maximal expression of parallelism. In contrast, the signals denoted by fields for which $k > 1$ are encoded, and *μinstructions* made up of such fields are referred to as *vertical μinstructions*. A benefit of such encoding is that the bit-width of *μinstructions* is significantly reduced. However, the parallelism which can be expressed through vertical *μinstructions* is limited and combinatory logic is needed to decode field values.

Orthogonal to the vertical/horizontal nature of a *μinstruction* is the architectural notion of how many *μoperations* a *μinstruction* can hold. If only a "few" *μoperations* can be placed into a *μinstruction* the machine has a *vertical architecture*; otherwise it has a *horizontal architecture*[8][3].

For architectures that support concurrent execution of *μoperations*, be they vertical architectures or horizontal architectures, the scheduling of *μoperations* presents an area of optimization. In this context, the goal of optimization is to produce a low-level *μprogram* having a minimal or near-minimal number or *μinstructions*. For this form of optimization, referred to as *μcode compaction*, achieving optimal results has been shown to be NP-complete[17]. There are two types of compaction: (1) *local compaction* which focuses on restructuring the *μoperations* within *straight-line μcode*(SLM) – also known as *basic blocks*, and (2) *global compaction* whose focus spans multiple SLMs.

III. Related Work

Research into the design of high-level *μprogramming* languages and *μcompilers* predominantly took place during the 1970's and early 1980's. A number of papers have been published on the topic of *μcode* optimization [5], [4], [9], [8]. Agerwala [1] has written a survey on *μcode* 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 *μprogram* 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 [1], 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 *μoperations*. This sequence of time frames is viewed as partitioning the computation defined by the high-level (input) *μprogram* 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 *μinstruction*.

SIMPL (Single Identity Microprogramming Language) [11] is a high-level (machine dependent) *μprogramming* 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 constitute an independent set of *μoperations*. 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 [7] 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 [12], 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 [2], 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.

IV. 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 *$\mathcal{P}$aradigm* compiler produces three artifacts to assist developers in their optimization-oriented analysis efforts: (1) views, (2) heat maps, and (3) estimation tables.

A. Views

$\mathcal{P}$aradigm 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.

$$\begin{tikzcd} \boxed{f} \arrow[r, "12"] & \boxed{g} \end{tikzcd}$$

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 *$\mathcal{P}$aradigm* compiler for a *μprogram* produced for a hypothetical machine.

B. Heat Maps

Heat maps are another form of feedback produced by the *$\mathcal{P}$aradigm* compiler. Specifically, the *$\mathcal{P}$aradigm* compiler will output twelve attributes to a file in a comma-separated value format. Attributes range from method arity, method size, reference_frequency, inlined - called size, to (inlined - 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.

C. Efficiency Estimator

In order to meet resource constraints, it may be necessary for developers to optimize their high-level *μprogram*. To facilitate optimization, *$\mathcal{P}$aradigm* provides high-level language

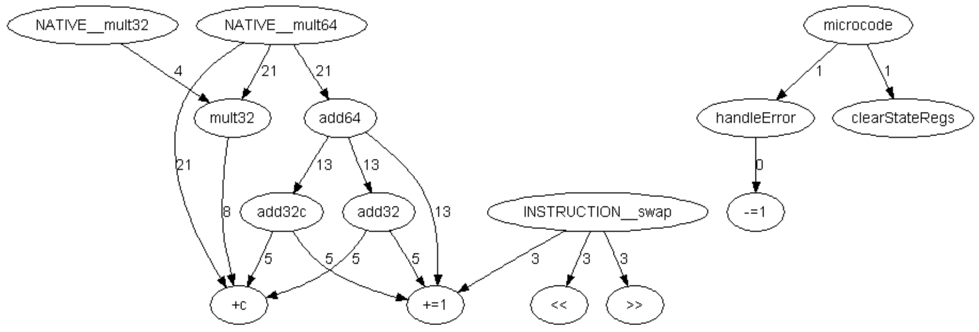

Figure 1. A view showing internal register allocations performed by the compiler.

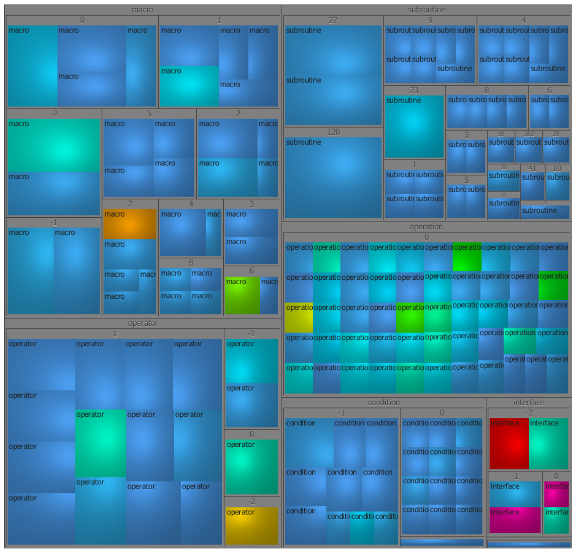

Figure 2. Heat map of *μcode* for a hypothetical machine.

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 *μcode* to expand dramatically. For example, suppose the body of a method m consists of 100 lines of *μcode*. Further suppose that m is called in 10 places in the *μcode*. If all 10 calls are in-lined, then in-lining (without compression) will yield 1000 *μinstructions*. In contrast, suppose that a *call* to the method m requires 20 lines of *μcode*. In this case, calling m 10 times will result in a total overhead of 200 lines of *μcode*. Thus, an implementation in which m is called will contain 700 fewer lines of *μcode*(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 *move* actual parameters to the internal registers corresponding to formal input parameters of the method. A *call* instruction must be generated by the compiler to transfer execution to the method body, and a *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 *μoperation* will fit into a *μinstruction*. Thus, going back to our previous example, if the execution of each *μinstruction* 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 $\mathcal{P}$*aradigm* compiler, provides an estimation of the effects of method call versus method in-lining. In particular, two sorted tables are produced: (1) a *static call-frequency estimation table*, and (2) an *execution path estimation table*. Examples describing the information in both of these tables are described in the sections that follow.

1) Example: Static Call-Frequency Estimation: Suppose method $m1$ is an in-line method candidate having 3 formal input parameters. Furthermore, let us assume that a static inspection of the $\mathcal{P}$*aradigm* application reveals that $m1$ is called from 10 syntactically distinct locations. Similarly, suppose method $m2$ is inline candidate method having 2 formal input parameters. Furthermore, let us assume that a static inspection of the $\mathcal{P}$*aradigm* application reveals that $m2$ is called from 20 syntactically distinct locations.

It should be noted that static-call-estimation provides a fairly course grained and basic estimation of the overhead associated with calling methods. In particular, static call

Static Call-Frequency Estimation

Method	Move Instruction Overhead	Static Overhead Sum
$m1$	$3 * 10 = 30$	$(3 + 2) * 10 = 50$
$m2$	$2 * 20 = 40$	$(2 + 2) * 20 = 80$

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 *µcode*. Note that in this example, there are only 2 lexical occurrences of $m1$. However, $m1$ will be called a total of $5 * 2 = 10$ 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.

2) Example: Execution Path Estimation: Suppose methods $m1$, $m2$ and $m3$ are respectively called 5, 6, and 4 times from the *µcode* 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: $1 * 5 + 1 * 6 * 2 + 1 * 4 * 3 * 2 = 41$. More specifically, $m1$ is called 5 times from the *µcode*. This accounts for the $1 * 5$ term. Next, $m1$ is called 2 times from $m2$, which itself is called 6 times from the *µcode*. This accounts for the term $1 * 6 * 2$. And finally, $m2$ is called 3 times from $m3$ which is called 4 times from the *µcode*. This accounts for the term $1 * 4 * 3 * 2$.

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 *µcode* 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:

$$100.0 - (inlined\_execution\_time / called\_execution\_time) * 100.00$$

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 *conditions*[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.

Method	Total Calls	Size	
		Inlined	Called
$m1$	41	0	83
$m2$	18	36	39
$m3$	4	12	12

Method	Time		% Speedup
	Inlined	Called	
$m1$	0	82	100%
$m2$	36	72	50.0%
$m3$	12	20	40.0%

Table I
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(); m2(); return();
}

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

Figure 3. Example used for execution path estimation

V. USER-DEFINED OPTIMIZATION RULES

The $\mathcal{P}$*aradigm* compiler is transformation-based and implemented in the TL System[15]. During compilation, a $\mathcal{P}$*aradigm* program is passed through a number of canonical forms, each of which can be output in human-readable form. The $\mathcal{P}$*aradigm*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 *µcode* design. Figure 4 is an example of a *µcode* fragment, which can be output by the compiler, consisting of a sequence of *µoperation* 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 $\mathcal{P}$*aradigm* has five different kinds of methods. The rational behind this is beyond the scope of this article.

```
label_f:  ...
writeReg(T1Type.SOME, AType.$temp_reg 3 );
copyReg( AType.$temp_reg 3 ,AType.$reg 2);
```

Figure 4. A μ*code* fragment prior to custom optimization.

hardware architecture together with our understanding of the semantics of the implementations of the *writeReg* and *copyReg* operations we conclude that the transformation shown in Figure 5 is correctness-preserving. Furthermore, suppose that additional analysis leads us to conclude that such a transformation would be correctness-preserving **in all contexts**. That is, regardless of how it gets generated by the compiler, whenever a "write" to a temp register X is followed by a "copy" from that temp register X to the register Y, then this pair of operations can be replaced by a single operation that will directly "write" to the register Y.

Given that these conditions hold, we would like to expand the functionality of the compiler to include such an optimizing transformation. $\mathcal{P}$*aradigm* supports such extension of its compiler through a special transformation module in which domain experts can place custom-designed program transformations. There are no restrictions on the nature of the transformations that can be created. In particular, optimizing transformations can be developed utilizing the full capabilities of the TL system.

```
writeReg(T1Type.SOME, AType.$temp_reg 3 );
copyReg( AType.$temp_reg 3 ,AType.$reg 2);

→

writeReg(T1Type.SOME,AType.$reg 2);
```

Figure 5. A custom program transformation.

VI. $\mathcal{P}$*aradigm*'s Timing Constraint Language

$\mathcal{P}$*aradigm* 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 μ*code* (SLM). Timing-constraint based optimization does not involve commutative reordering of μ*operations*, instead it focuses on maximizing the compression of adjacent (i.e., associative) μ*instructions*. It is worth mentioning that in the compilation stage where timing-constraint based optimization occurs, the μ*program* 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 μ*instructions* occurring between consecutive labels.

Conceptually, a timing constraint is a pair of logical formulas that, if satisfied by adjacent μ*instructions*, prevent them from being compressed into a single μ*instruction*. Compression is also (implicitly) prohibited in cases when corresponding fields, in adjacent μ*instructions*, 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, $\mathcal{F}_1$ and $\mathcal{F}_2$ 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 μ*instructions* $\mathcal{I}_j$ and $\mathcal{I}_{j+1}$ proceeds as follows: If $\mathcal{I}_j$ satisfies $\mathcal{F}_1$ and $\mathcal{I}_{j+1}$ satisfies $\mathcal{F}_2$, then we say that the the μ*instructions* $\mathcal{I}_j$ and $\mathcal{I}_{j+1}$ *satisfy* the timing constraint TC_k, in which case the compression of $\mathcal{I}_j$ and $\mathcal{I}_{j+1}$ is prohibited by TC_k; otherwise compression is not prohibited by TC_k.

```
constraint TC_k {

    first_row:  F_1;
    second_row: F_2;
}
```

Figure 6. An abstract example of a timing constraint.

TCL allows μ*code* compression to be restricted by a *set* of timing constraints $\mathcal{S}_{TC} = \{TC_1, \ldots, TC_m\}$. The compression of any pair of μ*instructions* $\mathcal{I}_j$ and $\mathcal{I}_{j+1}$ is prohibited if $\exists TC_k \in \mathcal{S}_{TC}$ such that TC_k is satisfied by the μ*instructions* $\mathcal{I}_j$ and $\mathcal{I}_{j+1}$.

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 μ*instructions* 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 $\mathcal{P}$*aradigm* compiler provides feedback summarizing the impact of the optimizations it performs. Figure II shows an example of an optimization summary.

VII. Conclusion

In the design of a high-level architecture-independent μ*programming* 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

```
Optimization Metrics:
Standard Compiler Optimizations.
Total number of temp register optimizations = 0
Number of nop() statements removed = 0

Custom Optimizations.
Total number of row reductions due to custom optimizations = 0

Constraint-based Optimizations.
Number of row mergings prevented due to timing constraints = 1291
Number of duplicate row mergings = 500
Number of conflict-free row mergings = 1000
Total number of constraint-based row mergings = 1500

Number of rows before any optimization = 2800
Number of rows after all optimization = 1300
Size of optimized file as a percentage of the unoptimized file = 46.43%
The size of the unoptimized file was reduced by = 53.57%
```

Table II
OPTIMIZATION FEEDBACK PROVIDED BY THE *Paradigm* COMPILER.

μcode 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

[1] T. Agerwala. Microprogram Optimization: A Survey. *Computers, IEEE Transactions on*, C-25(10):962–973, Oct. 1976.

[2] J. O. Bondi and P. D. Stigall. Designing HMO, an Integrated Hardware Microcode Optimizer. In *MICRO 7: Conference record of the 7th annual workshop on Microprogramming*, pages 268–276, New York, NY, USA, 1974. ACM.

[3] S. Dasgupta. The organization of microprogram stores. *ACM Comput. Surv.*, 11(1):39–65, Mar. 1979.

[4] S. Davidson, D. Landskov, B. Shriver, and P. Mallett. Some Experiments in Local Microcode Compaction for Horizontal Machines. *Computers, IEEE Transactions on*, C-30(7):460–477, July 1981.

[5] J. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *Computers, IEEE Transactions on*, C-30(7):478–490, July 1981.

[6] R. C. Haavind, Jr. The many faces of microprogramming: What started out as a convenience for systems designers may eventually bring computers much better tailored to users' needs. *SIGMICRO Newsl.*, 2(4):12–16, Jan. 1972.

[7] W. C. Hopkins, M. J. Horton, and C. S. Arnold. Target-Independent High-Level Microprogramming. In *MICRO 18: Proceedings of the 18th annual workshop on Microprogramming*, pages 137–144, New York, NY, USA, 1985. ACM.

[8] D. Landskov, S. Davidson, B. Shriver, and P. W. Mallett. Local microcode compaction techniques. *ACM Comput. Surv.*, 12(3):261–294, Sept. 1980.

[9] P. Marwedel. A Retargetable Compiler for a High-Level Microprogramming Language. *SIGMICRO Newsl.*, 15(4):267–274, 1984.

[10] J. A. McCoy. An Embedded System For Safe, Secure And Reliable Execution of High Consequence Software. In *Proceedings of the* 5^{th} *IEEE International Symposium on High Assurance Systems Engineering (HASE)*, pages 107–114. IEEE, 2000.

[11] C. Ramamoorthy and M. Tsuchiya. A High-Level Language for Horizontal Microprogramming. *Computers, IEEE Transactions on*, C-23(8):791–801, Aug. 1974.

[12] A. K. Tirrell. A Study of the Application of Compiler Techniques to the Generation of Micro-code. In *Proceedings of the meeting on SIGPLAN/SIGMICRO interface*, pages 67–85, New York, NY, USA, 1973. ACM.

[13] A. B. Tucker and M. J. Flynn. Dynamic microprogramming: Processor organization and programming. *Commun. ACM*, 14(4):240–250, Apr. 1971.

[14] M. V. Wilkes. The early british computer conferences. chapter The Best Way to Design an Automatic Calculating Machine, pages 182–184. MIT Press, Cambridge, MA, USA, 1989.

[15] V. L. Winter. Stack-based Strategic Control. In *Preproceedings of the Seventh International Workshop on Reduction Strategies in Rewriting and Programming*, June 2007.

[16] V. L. Winter, H. Siy, J. McCoy, B. Farkas, G. Wickstrom, D. Demming, J. Perry, and S. Srinivasan. Incorporating Standard Java Libraries into the Design of Embedded Systems. In K. Cai, editor, *Java in Academia and Research*. iConcept Press, 2011.

[17] S. S. Yau, A. C. Schowe, and M. Tsuchiya. On storage optimization of horizontal microprograms. In *Conference Record of the 7th Annual Workshop on Microprogramming*, MICRO 7, pages 98–106, New York, NY, USA, 1974. ACM.