A Timing Requirements-Aware Scratchpad Memory Allocation Scheme for a Precision Timed Architecture

Hiren D. Patel
Ben Lickly
Bas Burgers
Edward A. Lee

Electrical Engineering and Computer Sciences
University of California at Berkeley

Technical Report No. UCB/EECS-2008-115
http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-115.html

September 12, 2008

Copyright 2008, by the author(s).
All rights reserved.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission.

Acknowledgment

This work was supported in part by the Center for Hybrid and Embedded Software Systems (CHESS) at UC Berkeley, which receives support from the National Science Foundation (NSF awards #0720882 (CSR-EHS:PRET) and #0720841 (CSR-CPS)), the U. S. Army Research Office (ARO#W911NF-07-2-0019), the U. S. Air Force Office of Scientific Research (MURI #FA9550-06-0312), the Air Force Research Lab (AFRL), the State of California Micro Program, and the following companies: Agilent, Bosch, HSBC, Lockheed-Martin, National Instruments, and Toyota.

A Timing Requirements-Aware Scratchpad Memory Allocation Scheme for a Precision Timed Architecture

Hiren D. Patel, Ben Lickly and Edward A. Lee,
Department of Electrical Engineering and Computer Sciences,
University of California, Berkeley,
Berkeley, California 94720-1776
Email: {hiren,blickly,eal}@eecs.berkeley.edu

Bas Burgers
University of Twente,
Enschede, The Netherlands
Email: b.j.burgers@student.utwente.nl

September 12, 2008

Abstract

The precision timed architecture presents a real-time embedded processor with instruction-set extensions that provide precise timing control via timing instructions to the programmer. Programmers not only describe their functionality using C, but they can also prescribe timing requirements in the program. We target this architecture and present a static scratchpad memory allocation scheme that greedily attempts to meet these timing requirements. Our objective is to schedule minimum number of instructions and minimize data allocation to the scratchpads such that timing requirements in the program are met. Once the timing requirements are satisfied, the remainder of the scratchpad memory can be used to optimize some other metric desired by the programmer. As an example, we minimize the frequency of main memory accesses in the program. This work presents the following: 1) high-level timing constructs for C that synthesize to timing instructions and 2) a greedy iterative instruction and data scratchpad memory allocation scheme that attempts to first meet the specified timing requirements.

1 Introduction

In real-time embedded systems, scratchpad memories are often used as an alternative to on-chip cache memories. They offer a reduction in area, low power and low energy consumption solution. More importantly, they provide a high degree of timing predictability [2, 10, 18, 20]. Scratchpads do not require hardware policies to determine the transfers on and off the scratchpads, thus reducing hardware logic, power, area and improving timing predictability. But, the cost of using scratchpads is that the programmer is responsible for scheduling these transfers. Consequently, automatic compiler-level memory allocation schemes are desirable.

There are numerous scratchpad memory allocation schemes targeting non-real-time embedded systems [1, 12, 19] that focus on improving the average-case execution time (ACET) of a program. But simply improving the ACET

does not help in meeting the timing requirements of a real-time embedded application. Alternatively, worst-case execution time (WCET) based memory allocation schemes for real-time embedded systems are proposed in [17, 14, 4]. These approaches minimize a task's worst-case execution path by scheduling instructions and data words to the scratchpads. A real-time operating system (RTOS) and its scheduling algorithms are used to specify the programmer's timing requirements. They do not necessarily make an effort to meet the programmer's intended timing requirements because that information is unavailable in the program. Unfortunately, it is difficult for a compiler to do this because today's programming languages do not specify timing requirements and most embedded processors lack mechanisms to enforce them.

An exception is the precision timed (PRET) architecture proposed by Lickly et al [9]. Edwards and Lee [5] recently made the case to reintroduce time as a first-class property at all abstraction levels, and the PRET architecture [9] presents their initial efforts. This is a real-time embedded processor that provides timing as predictable as its logical function. It is a multithreaded processor based on the SPARC instruction-set architecture with scratchpad memories, extensions for precise timing control and no RTOS. It accepts C programs compiled with the GCC toolchain.

While using the PRET architecture, we noticed two issues that make its practical use difficult. They are: 1) timing requirements are encoded via low-level assembly commands and 2) there is an assumption that instructions and data fit entirely on the scratchpads. In our experience, it is inconvenient to use assembly-level instructions intermingled with C and any realistic application is typically greater than the scratchpad size. Our efforts in this work are to address these two issues.

Subprogram p_i

```
DEADFOR(500000)(;;) {
  DEADSEQ(3000) {
    inp = sample_input();
  };

  out = NULL;
  if (inp != NULL) {
    out = filter(inp);
    }
  DEADSEQ(10000) {
  if (output != NULL) {
    send_output(out);
    }
  };
};
```

Program Tests t_i

```
for(;;) {
DEADSTART0(500000,ad.c_main_3_xx500000xx_is_loop);
  DEADSTART1(3000,d.c_main_4_xx3000xx_);
    inp = sample_input();
  DEADEND1(ad.c_main_4_xx3000xx_);

  out = NULL;
  if (inp != NULL) {
    out = filter(inp);
    }
  DEADSTART1(10000 , ad.c_main_5_xx10000xx_);
  if (out != NULL) {
    send_output(out);
    }
  DEADEND1(ad.c_main_5_xx10000xx_);
}
DEADEND0(ad.c_main_3_xx500000xx_is_loop);
```

Command File c

```
skipfetch
lbreak 0 dma.c_main_3_xx500000xx_is_loop_start
lbreak 0 dma.c_main_3_xx500000xx_is_loop_end
lbreak 0 dma.c_main_3_xx500000xx_is_loop_end_end
lbreak 0 dma.c_main_4_xx3000xx_start
lbreak 0 dma.c_main_4_xx3000xx_end
lbreak 0 dma.c_main_4_xx3000xx_end_end
lbreak 0 dma.c_main_5_xx10000xx_start
lbreak 0 dma.c_main_5_xx10000xx_end
lbreak 0 dma.c_main_5_xx10000xx_end_end
profile
stats thread0-t0.pkl
quit
```

Figure 1: Simple Example using Timing Constructs

We address the first issue by introducing *timing constructs* in C with associated semantics. These synthesize to PRET's *timing instructions* [7, 5, 9] and they are used as a means of specifying *timing requirements* in the program. The synthesis of timing instructions is a source-to-source transformation for which we use an open-source C front-end

parser [3]. For the second issue, we propose a static memory allocation scheme that greedily assigns instructions and data words on the scratchpads to satisfy the timing requirements specified by the programmer.

Typically, scratchpad memory allocation schemes focused on improving ACET use profile-based methods, and schemes concerned about real-time (WCET-based) use static program analysis. The latter is particularly important for safety-critical and *hard* real-time embedded systems. But, we are interested in *soft* real-time embedded applications where timing violations as a result of not exercising the absolute worst-case path are undesired, but not hazardous. Furthermore, constantly evolving architectures and traditional input-data dependent programming styles hinder the wide use of static WCET analysis [15]. Since Lickly et al [9] clearly indicate that the PRET architecture is still in its infancy and evolving, we opt for a profile-based method. In the future, we plan to generate probability distributions from the profiled data and then perform the allocation with knowledge of the expected cases as well.

In this work, we leverage PRET's *timing instructions* [5, 9, 8] and introduce *timing constructs* for C as a means of specifying *timing requirements* in the program. We then shift our focus to a static memory allocation scheme that greedily assigns instructions and data words to scratchpads to satisfy the timing requirements and then stops, leaving resources available for non-real-time functions. Our objective is to schedule the minimum number of instructions and minimize data allocation to the scratchpads such that the timing requirements specified in the program are met. This allows the remaining scratchpad space to be used as the programmer sees fit, such as to minimize power consumption or the execution time. We present one possible approach that uses the remaining scratchpad space to minimize the frequency of main memory accesses of the entire program. We select a static memory allocation scheme approach because dynamic schemes add transfer instructions in the original program; possibly resulting in changing execution times. To our knowledge, this is the first scratchpad memory allocation scheme that tries to meet timing requirements specified in programs.

2 Related Work

Some work on scratchpad memory allocation schemes for non-real-time embedded applications are presented in [1, 12, 19, 13, 16, 21]. These approaches focus on improving ACET and investigate different criteria such as allocation schemes without fixed size memories, multiple heterogeneous memories in the memory hierarchy, an intersecting lifetime criterion for selecting arrays to allocate, reducing energy costs and so on. However, they are not concerned with real-time.

There are several works that focus on real-time embedded systems. Suhendra et al. [17] present a static WCET-based data allocation scheme for scratchpads using static program analysis. The approach is to iteratively minimize

the worst-case execution path in a task. Puaut [14] presents a dynamic WCET-based scratchpad instruction allocation scheme and later, Deverge and Puaut [4] extend this allocation scheme to data as well. None of these memory allocation schemes take into account timing requirements that may be specified by the user in the program and scheduling instructions and data to meet them.

3 Timing Instructions & Constructs

We begin by summarizing the timing instruction extensions for PRET called *deadline instructions*. Using these timing instructions, we define *deadline blocks* that are synthesized from our *timing constructs* used in C.

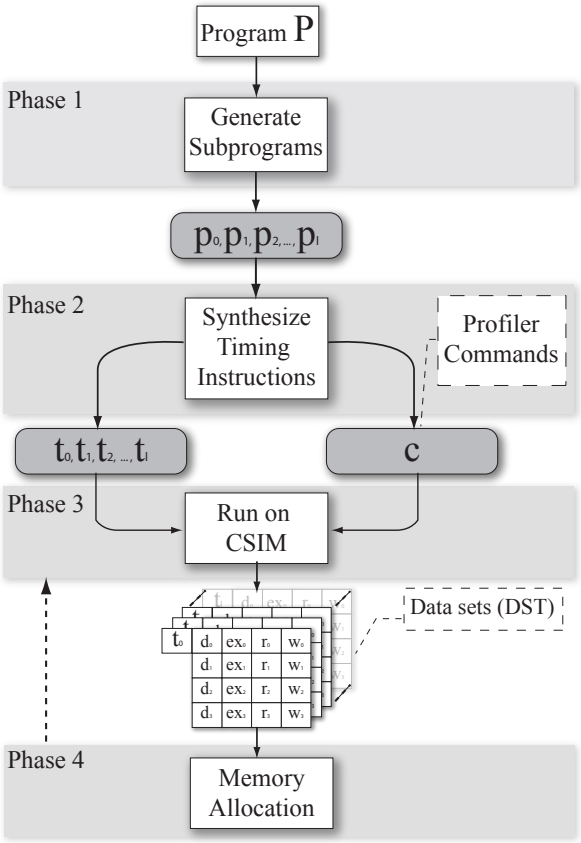

Figure 2: Memory Allocation Usage Flow

3.1 Deadline Instruction

The *deadline instruction* introduced by Ip and Edwards [7] is defined as a pair (dr_i, v) where dr_i is a deadline register and v is the value for the deadline ($0 \leq i \leq 7$). The semantics of the deadline instruction is summarized next. The

deadline instruction sets the deadline value v to the register dr_i if dr_i is zero and then decrements the value v every clock cycle. Note that the value in the deadline instruction represents a cycle timer. If however, dr_i is non-zero, the deadline instruction blocks until dr_i becomes zero, after which the new value v is loaded into the register. Additional implementation details of the PRET architecture are explained in [9].

3.2 Deadline Block

We define a *deadline block* d as a pair of deadline instructions $d = ((dr_i, v_0), (dr_i, 0))$. We call the first deadline instruction (dr_i, v_0) the start and the second $(dr_i, 0)$ the end. Notice that the end deadline instruction has a deadline register value 0. The program code that can be executed between a start and end deadline instruction is said to be contained within the deadline block. The value v_0 at the start of the deadline block specifies the *timing requirement* in cycles for the program code encapsulated. A *timing violation* of this requirement is when the code's execution within the deadline block takes longer to execute than the specified value. Within a deadline block, we allow most constructs such as function calls, `if-then`, `for` and `while`. However, we disallow control jumps that jump outside the deadline block such as `break`, `continue`, `return` and `goto`. Note that a deadline block never finishes earlier than its specified value v_0. This is a key part in achieving *repeatable execution* times from the architecture. The program's execution stalls if the execution reaches the end of a deadline block before v_0 reaches 0 and it resumes once v_0 is 0.

3.3 Timing Constructs

We introduce two broad categories of *timing constructs*: sequential and loop. The sequential constructs (`DEADSEQ()`) enforce a lower-bound on the execution time on a sequential segment of program code. On the other hand, the loop constructs (`DEADFOR(), DEADWHILE(), ...`) enforce a periodic execution of the code within the scope. Figure 1 shows an example of the use of timing constructs and the result after the transformation. Note that the arguments after the deadline register values are unique labels used for profiling. Also note that we have specifically defined the synthesis of these instructions to match the appropriate semantics. Currently, we encode the deadline register in cycles as well, but at the same time we are investigating the specification of timing requirements in time and frequency units that are again automatically synthesized to cycles using a front-end parser.

4 Profiler and Memory Allocation Usage Flow

Figure 2 presents a block-level diagram of the phases involved in profiling information about a program P. Program P is a C program with timing constructs. Note that these constructs are currently only supported by the PRET architecture.

`Phase 1` receives a program P as an input and produces multiple subprograms p_i. The purpose of these subprograms is to exercise different paths in the program flow of the original program P. The subprograms can be either generated manually or via third-party test generation tools. Currently, we manually create the subprograms.

`Phase 2` takes these subprograms and synthesizes timing instructions from the timing constructs resulting in program tests t_i. In addition, we automatically annotate the source with profiling labels. The profiler identifies the start and end of deadline blocks via these labels. A profiler command file c is generated during this phase. We use the open-source C/C++ front-end Clang [3] to implement the necessary rules to conduct the source-to-source transformations. We also automatically allocate deadline registers to the timing constructs. Figure 1 shows a simple example. By performing source-to-source transformations we do not have to change the implementation of the C standard implemented in GCC for these new timing constructs.

The actual profiling is done in `Phase 3`. The program tests and the command file are input to the profiler $CSIM$. The output of the profiler are data sets DST for every test t_i. The information these data sets contain are:

- For every deadline block: Execution time, instructions encountered and their frequency, data words read and written to and their frequency.
- Other blocks: instructions encountered and their frequency, data words read and written to and their frequency, total execution time of test.

The final `Phase 4` is where the memory allocation is done. The first part of the memory allocation is a feasibility check that ensures that a memory allocation that results in meeting the timing requirements does exist for some scratchpad size. If one does exist, then a greedy algorithm is used to allocate instructions and data words to the scratchpads. Note that there is a back arrow to `Phase 3` because after an allocation is performed, the profiling is rerun. This process continues until `Phase 4` yields an allocation that meets the specified timing requirements or fails due to scratchpad size.

5 Static Memory Allocation

We pictorially represent our allocation algorithm in Figure 3 and describe the stages briefly. Note that Table 1 is a list of variables we employ throughout the remainder of the paper.

T	set of tests.
$t \in T$	one test in T.
$INST(t)$	set of instructions encountered in test t.
$DATA(t)$	set of data words written to/read from in test t.
$D(t)$	set of deadline blocks in test t.
$d \in D(t)$	a deadline block.
$X(d)$	set of instructions executed in deadline block d.
X	set of instructions executed in any deadline block.
$Y(d)$	set of non-global data words read from or written to in deadline block d.
Y	set of data read or written in any deadline block.
$SIZE(m)$	size of memory unit m.
$\Delta_{mm}(d)$	cycles missed for deadline block d when in main memory.
$N(x)$	frequency of instruction x encountered.
$N(x, d)$	frequency of instruction x encountered in deadline block d.
$N_r(y)$	frequency of reading data word y.
$N_r(y, d)$	frequency of reading data word y in deadline block d.
$N_w(y)$	frequency of writing data word y.
$N_w(y, d)$	frequency of writing data word y in deadline block d.
A_{mm}	accessing time to main memory via the memory wheel.
$ALLOC$	number of words to allocate to scratchpad at each iteration.
PRE_{inst}	set of previously scheduled instructions.
PRE_{data}	set of previously scheduled data words.

Table 1: Variable Definitions

5.1 Feasibility Check

Before performing any allocation, we first perform a feasibility check. We say that a program P is infeasible if it cannot meet all timing requirements for every program test t_i. In order to check for infeasibility, we assume that all instructions and data are in a scratchpad, and run each of the tests. The feasibility check is successful if none of the timing requirements are violated. Otherwise, the program is infeasible and the user must alter the timing requirements in the original program.

5.2 Memory Allocation

The memory allocation in `Phase 4` is refined in Figure 3. This step is reached only after the feasibility check passes. We begin by identifying deadline blocks that violate its timing requirements. If there are violations but the existing

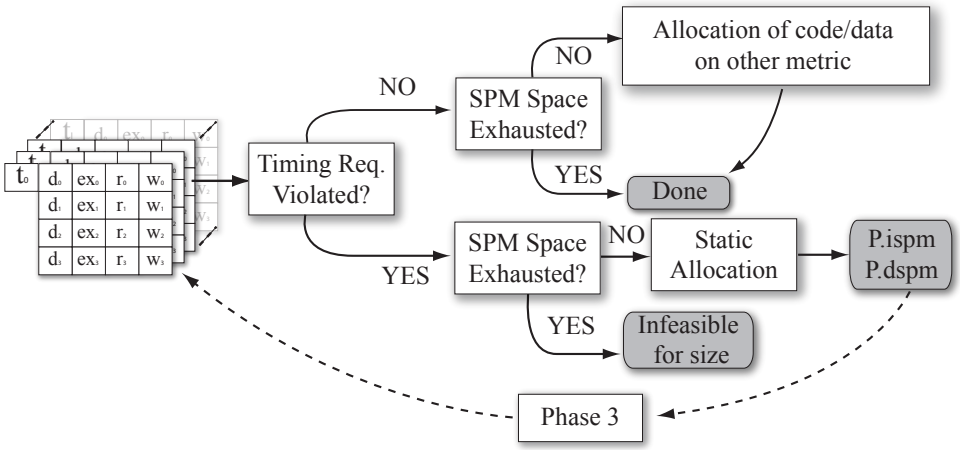

Figure 3: Iterative Static Allocation Flow

allocation has occupied the maximum size of the scratchpad memories, then for the given scratchpad memory size it is not possible to meet the timing requirements. On the other hand, if there is space available on the scratchpads, we perform instruction and data allocation and merge the existing allocation with the generated allocation. Note that once an allocation is generated, those instructions and data words are not reconsidered for allocation during the following iterations. If there are no deadline blocks that missed their deadlines, then we have successfully found an allocation that meets all timing requirements. In this case, we can use any remaining space on the scratchpad as we see fit, including for scheduling instructions and data encountered outside of the deadline blocks. In our implementation, we allocate on the basis of the frequency of main memory accesses in the whole program, but any metric could be used.

5.3 Previously Scheduled Instructions and Data

The sets PRE_{inst} and PRE_{data} hold all the instructions and data allocated during the previous iterations. We allocate these instructions and data words at the beginning of each iteration and remove them from the set of instructions and data considered for memory allocation.

5.4 Identify Violating Deadline Blocks

We construct a set of deadline blocks V that violate their timing requirements. However, it is possible for a deadline block to violate its timing requirements in more than one test. It may also be the case that the program paths taken in the tests are different from each other; thus, different instructions and data words are encountered during profiling. As a result, we use a simple heuristic to collate the instructions, data words and execution times of these deadline blocks. For all tests where the deadline block missed its deadline, we select the maximum execution time of the deadline block

and the union of both instructions and data. These updated deadline blocks are added in set V and their corresponding information is merged as described above. Therefore, any indexing based on deadline blocks in V yields the merged information.

5.5 Exhausting Scratchpad Space

At every iteration we check if the remaining space in the scratchpad allows for scheduling $ALLOC$ addition words of instructions or data. In the event that the remaining space is less than $ALLOC$, we alter the variable size to that of the remaining space. If however, there is no space left at all on the scratchpad and timing requirements are violated, the allocation fails. Otherwise, if there are no timing violations, the allocation terminates successfully as shown in Figure 3.

5.6 Instruction

An access to the main memory goes through the memory wheel [9] that can take at worst A_{mm} cycles to complete. For a deadline block d, the memory access cost for an instruction $x \in X(d) \backslash PRE_{inst}$ is no more than the frequency of that instruction $N(x, d)$ multiplied by the main memory access time A_{mm}.

We formulate a binary integer linear programming problem for scheduling both instructions and data encountered in deadline blocks. We will start by defining the variables that correspond to instructions that could be added to the scratchpads.

$$I_{inst}(x) = \begin{cases} 1 & \text{allocate instruction } x \text{ to scratchpad.} \\ 0 & \text{otherwise.} \end{cases}$$

The instruction savings $IS(V)$ is given below. Instructions that contribute most to the instruction access time from main memory are more heavily weighted than those that do not.

$$IS(V) = \sum_{d \in V} \sum_{x \in X(d) \backslash PRE_{inst}} I_{inst}(x) N(x, d) A_{mm}$$

5.7 Data

Once again, we are interested in scheduling data access that contribute most to the cost in memory access time. Due to the simplicity of the PRET architecture, accesses for data to main memory also go through the memory wheel with worst-case access time of A_{mm}. For a deadline block d, a data word $y \in Y(d) \backslash PRE_{data}$ is written to $N_w(y, d)$ and read from $N_r(y, d)$ number of times. Therefore, the memory access cost for y is the memory access time A_{mm}

multiplied by the sum of the number of times y was read from and written to in deadline block d.

$$I_{data}(y) = \begin{cases} 1 & \text{allocate data word } y \text{ to scratchpad} \\ 0 & \text{otherwise.} \end{cases}$$

The savings from allocating data words $DS(V)$ is highest when data words that contribute most to the main memory access cost are allocated.

$$DS(V) = \sum_{d \in V} \sum_{y \in Y(d) \setminus PRE_{data}} I_{data}(y) A_{mm} \Big[N_r(y,d) + N_w(y,d) \Big]$$

5.8 Allocation

Our binary integer linear program shown below maximizes the savings occupied by either instruction or data $S(v)$. Thus, the objective function that we maximize is the sum of the savings from each of the instruction and data allocations:

Maximize:

$$S(V) = IS(V) + DS(V)$$

The constraint equations must make sure that we do not allocate more words than we are allowed in any given iteration. Remember that we keep history of the previously allocated instructions and merge the new ones with the previous ones.

Subject to:

$$\Big[\sum_{x \in X \setminus PRE_{inst}} I_{inst}(x) \; + \sum_{y \in Y \setminus PRE_{data}} I_{data}(y) \Big] \leq ALLOC$$

Solving this binary integer linear program gives us the allocation of data and instructions to the scratchpads in a single iteration of our allocation scheme. These iterations continue as shown in Figure 3 modifying PRE_{inst} and PRE_{data} to reflect the contents added to the scratchpad in each iteration.

5.9 Minimize Frequency of Main Memory Accesses

After all deadline blocks meet the timing requirements, one final iteration takes place to schedule the remaining scratchpad space. We decide to minimize the frequency of main memory accesses in the entire program, but other optimizations for low power and energy are also possible. We define $INST$ and $DATA$ as the remaining set of instructions and data words, respectively.

$$INST = \bigcup_{t \in T} INST(t) \backslash PRE_{inst}$$
$$DATA = \bigcup_{t \in T} DATA(t) \backslash PRE_{data}$$

We redefine the binary variables for the instruction and data.

$$I'_{inst}(x) = \begin{cases} 1 & \text{allocate instruction } x \text{ to scratchpad in final phase.} \\ 0 & \text{otherwise.} \end{cases}$$

$$I'_{data}(y) = \begin{cases} 1 & \text{allocate data word } y \text{ to scratchpad in final phase.} \\ 0 & \text{otherwise.} \end{cases}$$

We select the instructions and data words that cost the most in terms of frequency of main memory accesses by solving another binary integer linear program. In this case, our objective function is $S'(INST, DATA)$.

Maximize:

$$S'(INST, DATA) = \sum_{x \in INST} I'_{inst}(x) N(x) + \sum_{y \in DATA} I'_{data}(y) \Big[N_r(y) + N_w(y) \Big]$$

During this iteration, we want to allocate all of the remaining space on the scratchpad, which we describe in the constraint below.

Subject to:

$$\sum_{x \in INST} I'_{inst}(y) \leq SIZE(spm) - SIZE(PRE_{inst})$$

$$\sum_{y \in DATA} I'_{data}(y) \leq SIZE(spm) - SIZE(PRE_{data})$$

6 Experiments

Our infrastructure uses Clang [3] to perform the source-to-source transformations from timing constructs in C to timing instructions compilable by SPARC's GCC toolchain. We use Python for the profiling and integration with PRET's cycle-accurate simulator and `lp_solve` as the solver for selecting the instructions and data words that offer the most savings in terms of main memory access time. We select a few C benchmarks from the Malardalen [6] suite and augment them with timing requirements.

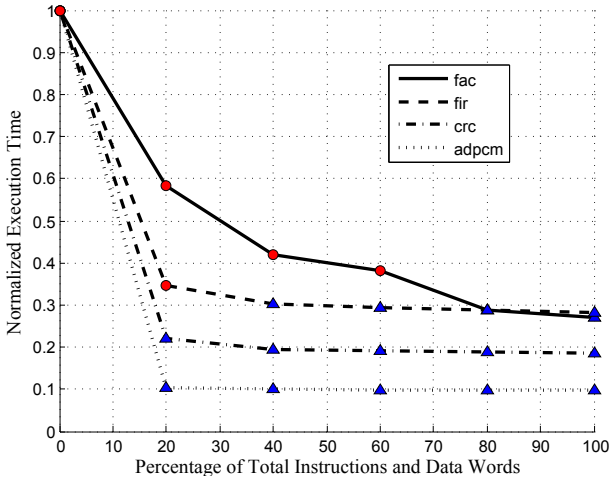

Figure 4: Execution Times with Varying Scratchpad Size

In Figure 4, we vary the scratchpad size (both instruction and data) as a percentage of the program instructions and data, and present the execution times of the benchmarks. At `0%` only the main memory is used. We also denote whether allocation failed at each of the scratchpad sizes; a circle denotes failure and a triangle denotes success. As shown, the selected benchmarks eventually met their timing requirements. In addition to the benchmarks shown in Figure 4, we plan to port segments of real-time programs from PapaBench [11].

7 Conclusion

The PRET architecture makes the execution time of programs repeatable through its timing instruction extensions. It also allows the programmers to specify their timing requirements within their programs, unlike traditional embedded programming languages like C. In this work, we leverage these timing instructions and provide high-level timing constructs to specify timing requirements in the program. Afterward, we present a static scratchpad memory allocation scheme that focuses on meeting these timing requirements in the program. This effort removes the assumption made by Lickly et al. [9] of the program and data having to fit entirely on the scratchpads. In our approach, a programmer specifies these timing requirements in the program using timing constructs. We automatically synthesize timing instructions from these timing constructs and additional profiling annotations. We iteratively schedule instructions and data words to the scratchpads for blocks that specify the programmer's timing requirements such that the least amount of scratchpad space is used to meet the timing requirements. If and once they are satisfied, any remaining space can be used to optimize based on another metric, such as reducing power or energy consumption.

References

[1] Oren Avissar, Rajeev Barua, and Dave Stewart, *An optimal memory allocation scheme for scratch-pad-based embedded systems*, ACM Transactions on Embedded Computing Systems **1** (2002), no. 1, 6–26.

[2] Rajeshwari Banakar, Stefan Steinke, Bo-Sik Lee, M. Balakrishnan, and Peter Marwedel, *Scratchpad memory: design alternative for cache on-chip memory in embedded systems*, CODES '02: Proceedings of the tenth international symposium on Hardware/software codesign (New York, NY, USA), ACM, 2002, pp. 73–78.

[3] clang Community, *clang: a C language family front-end for LLVM*, Website: http://www.clang.llvm.org.

[4] Jean-Francois Deverge and Isabelle Puaut, *WCET-Directed Dynamic Scratchpad Memory Allocation of Data*, Proceedings of the 19th Euromicro Conference on Real-Time Systems (2007), 179–190.

[5] Stephen A. Edwards and Edward A. Lee, *The case for the precision timed (PRET) machine*, June 2007, pp. 264–265.

[6] Jan Gustafsson, *The worst-case execution time tool challenge 2006*, Proc. 2 ndInternational Symposium on Leveraging Applications of Formal Methods (06), November (2006).

[7] Nicholas Jun Hao Ip and Stephen A. Edwards, *A processor extension for cycle-accurate real-time software*, Proceedings of the IFIP International Conference on Embedded and Ubiquitous Computing (EUC) (Seoul, Korea), vol. 4096, August 2006, pp. 449–458.

[8] Insup Lee, Susan Davidson, and Victor Wolfe, *Motivating time as a first class entity*, Technical Report MS-CIS-87-54, Dept. of Comp. and Infor. Science, Univ. of Penn, Aug. (Revised Oct.) 1987, Taxonomy of timing properties that must be expressible.

[9] Ben Lickly, Isaac Liu, Sungjun Kim, Hiren D. Patel, Stephen A. Edwards, and Edward A. Lee, *Predictable programming on a precision timed architecture*, Tech. Report UCB/EECS-2008-40, EECS Department, University of California, Berkeley, Apr 2008, This paper has been accepted for publication at CASES 2008.

[10] Peter Marwedel, Lars Wehmeyer, Manish Verma, Stefan Steinke, and Urs Helmig, *Fast, predictable and low energy memory references through architecture-aware compilation*, ASP-DAC '04: Proceedings of the 2004 conference on Asia South Pacific design automation (Piscataway, NJ, USA), IEEE Press, 2004.

[11] Fadia Nemer, Hugues Cassé, Pascal Sainrat, Jean-Paul Bahsoun, and Marianne de Mischiel, *Papabench: A Free Real-time Benchmark*, 6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis, Dresden, Germany (2006).

[12] Nghi Nguyen, Angel Dominguez, and Rajeev Barua, *Memory allocation for embedded systems with a compile-time-unknown scratch-pad size*, Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems (2005), 115–125.

[13] Preeti Ranjan Panda, Nikil D. Dutt, and Alexandru Nicolau, *Efficient utilization of scratch-pad memory in embedded processor applications*, European Design and Test Conference, 1997. ED&TC 97. Proceedings (17-20 Mar 1997), 7–11.

[14] Isabelle Puaut, *WCET-centric software-controlled instruction caches for hard real-time systems*, Proceedings of the 18th Euromicro Conference on Real-Time Systems (2006), 217–226.

[15] Peter Puschner, *Is worst-case execution-time analysis a non-problem? – towards new software and hardware architectures*, Proc. 2nd Euromicro International Workshop on WCET Analysis (York YO10 5DD, United Kingdom), Technical Report, Department of Computer Science, University of York, Jun. 2002.

[16] S. Steinke, L. Wehmeyer, Bo-Sik Lee, and P. Marwedel, *Assigning program and data objects to scratchpad for energy reduction*, Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings (2002), 409–415.

[17] Vivy Suhendra, Tulika Mitra, Abhik Roychoudhury, and Ting Chen, *WCET centric data allocation to scratchpad memory*, 05: Proceedings of the 26th IEEE International Real-Time Systems Symposium (2005).

[18] Lothar Thiele and Reinhard Wilhelm, *Design for timing predictability*, Real-Time Syst. **28** (2004), no. 2-3, 157–177.

[19] S. Udayakumaran and R. Barua, *An integrated scratch-pad allocator for affine and non-affine code*, Proceedings of the conference on Design, automation and test in Europe: Proceedings (2006), 925–930.

[20] Lars Wehmeyer and Peter Marwedel, *Influence of onchip Scratchpad Memories on WCET Prediction*, Proceedings of the 4th International Workshop on Worst-Case Execution Time (WCET) Analysis (2004).

[21] Liping Xue, Mahmut Kandemir, Guangyu Chen, and Taylan Yemliha, *Spm conscious loop scheduling for embedded chip multiprocessors*, (2006), 391–400.