Limitations of Adaptable System Architectures for WCET Reduction

Jack Whitham and Neil Audsley
Real-Time Systems Group
Department of Computer Science
University of York, York, YO10 5DD, UK
jack@cs.york.ac.uk

Abstract

This paper identifies three major issues facing worst-case execution time (WCET) reduction algorithms on adaptable architectures based on research carried out for the MCGREP-2 CPU project. The issues are exposing more instruction level parallelism (ILP) in code, reducing loading costs for the memory and processing elements used to reduce WCET, and making use of application-specific hardware. Potential difficulties in each of these areas are identified and possible solutions are proposed.

1 Introduction

Embedded systems often include some real-time functionality, such as control of external machinery [1]. Real-time tasks must operate within known time bounds (*deadlines*) in order for the overall system to be *safe*, and this poses an additional requirement for software design. Computing the *worst-case execution time* (WCET) of real-time tasks is an important step towards assuring the safety of the overall *real-time system* (RTS) [18]. WCET *reduction* for a task is a closely related problem involving the allocation of some memory or computing resource in order to minimize the WCET (Figure 1).

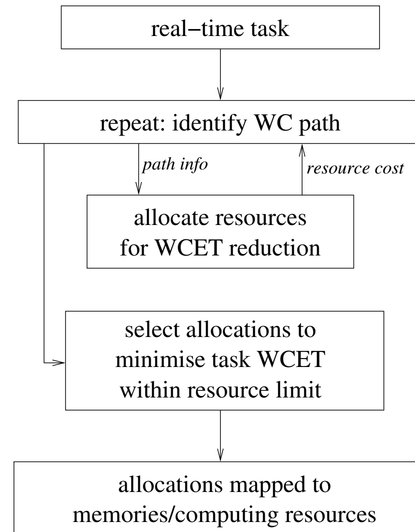

Figure 1. Generalized WCET reduction process.

General execution speed-up technologies such as cache memory, deep CPU pipelines and out-of-order superscalar issue units [15] are good for *average case execution time* (ACET) reduction, but the dynamic behavior of these components makes computing the WCET more difficult [10,24]. Therefore, even if the WCET of a task can be reduced by such techniques, the safety of the RTS cannot be easily assured. This motivates approaches that explicitly reduce the WCET of programs without introducing dynamic behavior, either automatically or with programmer assistance [28]. Automatic approaches have their roots in hardware/software co-design, i.e. partitioning tasks and subtasks between hardware and software in order to meet an optimization goal [2], e.g. ACET or WCET minimization. However, because of the relative difficulty of evaluating the resource consumption of candidate partitions (requiring hardware synthesis [8]) and because of the differences between hardware and software languages [7], current WCET reduction techniques avoid any need to generate hardware and instead operate by migrating subtasks into memory units that enable faster execution. These include instruction scratchpads [17] and lockable caches [3]. In these cases, the partitioning problem is relatively simple and can be solved by fast heuristics [21].

Some forms of adaptive system facilitate WCET reduction. The MCGREP-2 CPU [25–27] provides a *writable control store* (WCS) [20] that can store subtasks encoded as microinstructions (Figure 2). Our recent work [28] shows that subtasks can be selected from the *worst-case execution path* (WC path) of a program and translated automatically into microinstructions: this leads to greater WCET reductions than instruction scratchpad techniques because *instruction level parallelism* (ILP) can be exploited to execute the WC path in a shorter time period. MCGREP-2 is adaptive and reconfigurable in the sense that the control store can be updated at any time, allowing an unlimited number of tasks to benefit from WC path optimizations. Using microinstructions to implement subtasks is predictable in two senses: (1) execution timings are not data-dependent, and (2) resource consumption is easily computed [28]. However, the speed of each subtask is limited by the microarchitecture and the input program.

This paper explores the issues that limit WCET re-

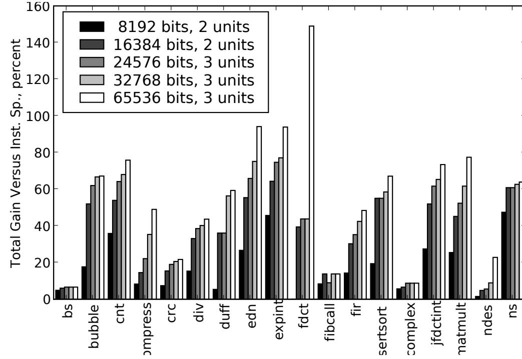

Figure 3. WCET reductions with various MCGREP-2 configurations using various benchmark programs.

duction within a general adaptive reconfigurable system, based on our research for the MCGREP-2 CPU project and its associated WCET reduction algorithms [25, 28]. We assume that WCET reduction begins with a task specified in a software language such as C, and then proceeds through a fully automatic process described in previous work [17,21,28] (such processes follow the general outline of Figure 1). We consider WCET reduction algorithms making use of scratchpads, locked caches, co-processors and run-time reconfigurable hardware [11], such as a *field programmable gate array* (FPGA).

Although WCET reductions can be achieved using the MCGREP-2 WCS, the execution time improvements that have been demonstrated are currently limited to about 50-150% [25] over an instruction scratchpad [17, 21] (Figure 3). Independent of the architecture actually used to implement the WCS [27], and independent of the technology used to apply WCET reductions [28], the magnitude of the possible reduction is limited by three major factors. These are: (1) the ILP available within the task, (2) the cost of loading the control store with the required information, and (3) the speed of the general-purpose microarchitecture that executes the microinstructions. These factors apply to any WCET reduction process, whether it is based on a scratchpad, locked cache, or some form of run-time reconfigurable hardware. The issues related to each are examined in sections 2, 3 and 4. Section 5 concludes.

2 The ILP Limitation

The ILP available within each task is influenced by both the source code and the compiler. WCET reduction approaches that operate by allocating instruction scratchpad or lockable cache space [3, 17, 21] do not consider ILP in code since conventional machine instructions are sequential. Trace scratchpad allocation approaches [28] do consider ILP, as machine instructions are converted into explicitly parallel code for storage in the WCS. This simplifies WCET analysis, but also implies that the degree of WCET reduction is limited by the ILP in the task.

In many programs, the degree of ILP is limited to two or three instructions within a single basic block, and around twice that number if basic block boundaries can be ignored through speculation [22]. This is a hard limit on WCET reduction for general software. For ACET reduction, the limit is approached by current superscalar CPU designs, and some of the same principles can be applied for WCET reduction [28]. Reaching this limit is an implementation challenge requiring the design of superscalar CPU pipelines that are also amenable to timing analysis.

Obtaining WCET reductions *beyond* the ILP limit is a language issue. Tasks that are *vectorisable* can be parallelised across a very large number of processing elements [22], because most subtasks are independent of each other. However, not all programming languages allow vectorisable code to be declared. A limited form of automatic vectorization is provided by *modulo scheduling* [4], but in general a specialist language is needed. The compiler needs additional information about data and control dependences in order to be able to arrange the subtasks for vector processing. Dataflow languages provide the required features, allowing both coarse-grained [5, 9] and fine-grained [30] reconfigurable arrays to be programmed. More conventional languages can also support extensions for vectorization, e.g. [12].

3 The Load Cost Limitation

Regardless of whether WCET reductions are provided by a scratchpad, locked cache or run-time reconfigurable hardware, a loading time cost is incurred whenever the configuration is updated. Some WCET reduction algorithms assume that loading takes place before task start-up [21, 28], but this is restrictive because it places a limit on the complexity of each task. This limit also applies to WCET reduction approaches that make use of fixed co-processors, since these are not run-time reconfigurable. The solution is to allow loading during execution and incorporate it into the WCET reduction process [17], after partitioning each task into regions with local memory maps [16]. The total loading time must be less than the total WCET reduction that is achieved.

Loading costs for scratchpads and locked caches are small, since burst-mode transfers can be used to rapidly move information from large external memory into smaller scratchpads. In a task with sufficient temporal locality on the WC path (e.g. many loops), loading time will be significantly smaller than the WCET for both instruction scratchpads [17] and a WCS [25]. However, the degree of temporal locality that is required is higher. If one instruction can be loaded into an instruction scratchpad in a single clock cycle, and then executed in a single clock cycle, then that instruction only needs to be executed twice to recover the cost of loading the scratchpad. But microinstructions are often larger than conventional instructions, and consequently more executions are required to recover

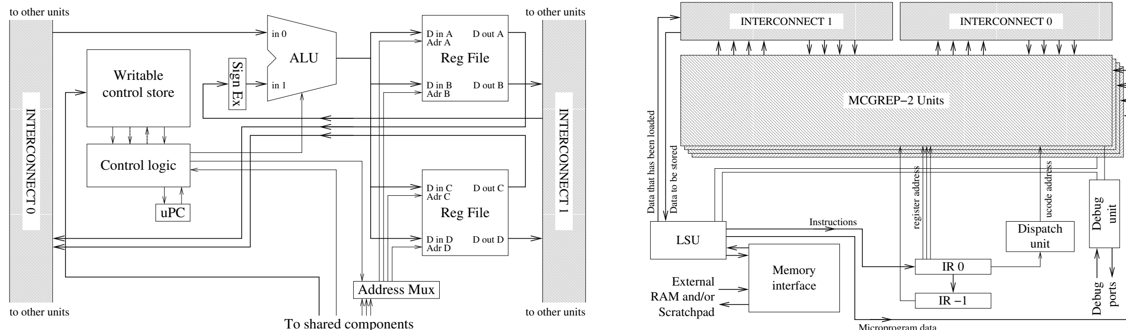

Figure 2. MCGREP-2 CPU: one array unit (left) and top level (right). MCGREP-2 is a simple form of coarse grained reconfigurable architecture (CGRA) in which each array unit is a small CPU capable of executing code from a writable control store, which can be used to reduce task WCETs.

the loading cost.

Loading can be carried out in parallel with task execution by introducing a *direct memory access* (DMA) controller to manage the copying process. A second task may be executed during the loading process, or the first task may continue execution as information is loaded into scratchpad for use in the near future. Both techniques have been previously explored by research into *overlaying* [14] and are implemented by modern CPU architectures in the form of *simultaneous multithreading* (SMT) and cache filling. Predictable forms of these dynamic operations could be used to eliminate effective loading costs in some cases.

We believe that loading costs are likely to become a significant problem for some systems. Run-time reconfigurable hardware loading costs can be very large: typical FPGA bitstream sizes are given by [30]. Run-time reconfigurable systems may include decompression modules to reduce the cost [11]. Consequently, a very high degree of temporal locality is required to reduce the overall WCET unless loading costs can be eliminated by parallel operation.

4 The General Purpose CPU Architecture Limitation

Perhaps the most serious limitation of present WCET reduction approaches is the assumption that a general purpose architecture is used. A conventional general-purpose CPU is assumed by instruction scratchpad and locked cache allocation approaches [3, 17, 21]. Although the MCGREP-2 CPU is extensible with *application-specific instruction set processor* (ASIP) features [6], such as custom instructions to accelerate WC path execution, these must be declared and applied explicitly by the programmer. Automatic WCET reduction algorithms for scratchpads only make use of standard ALU features at present.

CGRA architectures [5, 9] provide arrays that are specialized for vectorisable code. Mapping subtasks to such architectures is one way to reduce WCET, but large CGRAs are not suitable for general programs because insufficient ILP is available. The same problem applies to fine-grained arrays such as FPGAs, but these can provide even greater reductions because the logic gates can be specialized to a particular task. Automatic ASIP custom instruction selection for WCET reduction has been explored [31], and it is known that large ACET [13] and WCET [23] reductions are possible by migrating software into FPGA hardware, even without vectorisable code. Since run-time reconfiguration can be used to load task-specific hardware, customized hardware could be used in a similar manner to an instruction scratchpad or WCS, but with greater potential WCET reductions than either approach.

However, the search algorithms used to find the best allocations for WCET reduction become far more complex, since it is not easy to calculate the resource consumption of each allocation decision. To get an exact answer, a complete FPGA or ASIC synthesis process must be executed with all chosen components in place, and this is computationally expensive. Estimation is commonly used instead [29, 31], but this lowers the accuracy of allocation decisions and is likely to lead to poor utilization of space, or backtracking in the event of an overestimate. Scratchpad allocation algorithms can make use of all available space [17, 28] because exact computation of resource usage is very fast, and backtracking is not necessary [21].

We believe that WCET reduction using custom hardware is a form of co-design problem, and therefore NP-hard [19]. However, with appropriate restrictions and assumptions, WCET reduction can nevertheless be applied effectively using custom hardware. For example, run-time reconfigurable modules (e.g. [11]) of fixed size could be generated to provide WCET reductions to specific subtasks: this would isolate the WCET reduction process from the considerations of resource consumption and on-chip communication.

5 Conclusion

This paper has explored three issues that affect the degree of WCET reduction available for tasks in an adaptable architecture. Major challenges exist: specifying vectorization in order to exploit greater ILP is important [22], as is minimizing loading time costs [17]. Finding a way to apply WCET reduction algorithms to custom hardware may be the most rewarding challenge, as large execution time reductions are possible [13,23] if the technical issues of efficiently searching for the best resource allocation can be solved. These problems have been given only partial consideration by existing work. Solutions would allow embedded real-time systems to carry out more operations per time unit by explicitly reducing the WCET of each task.

References

[1] A. Burns and A. J. Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, 2001.

[2] R. Ernst, J. Henkel, and T. Benner. Hardware-software cosynthesis for microcontrollers. *IEEE Des. Test*, 10(4):64–75, 1993.

[3] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proc. CODES+ISSS*, pages 143–148, New York, NY, USA, 2007. ACM Press.

[4] J. Fisher, P. Faraboschi, and C. Young. *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. Morgan Kaufmann, 2004.

[5] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R. R. Taylor. PipeRench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[6] R. E. Gonzalez. Xtensa — A configurable and extensible processor. *IEEE Micro*, 20(2):60–70, 2000.

[7] B. Grattan, G. Stitt, and F. Vahid. Codesign-extended applications. In *Proc. 10th Int. Symp. Hardware/Software Codesign*, pages 1–6, 2002.

[8] R. K. Gupta and G. D. Micheli. Hardware-software cosynthesis for digital systems. *IEEE Des. Test*, 10(3):29–41, 1993.

[9] R. Hartenstein, M. Herz, T. Hoffmann, and U. Nageldinger. KressArray Xplorer: a new CAD environment to optimize reconfigurable datapath array. In *Proc. ASP-DAC*, pages 163–168, New York, NY, USA, 2000. ACM Press.

[10] R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proc. IEEE*, 91(7):1038–1054, 2003.

[11] M. Hubner and J. Becker. Exploiting dynamic and partial reconfiguration for FPGAs: toolflow, architecture and system integration. In *Proc. SBCCI*, pages 1–4, New York, NY, USA, 2006. ACM Press.

[12] Intel. Optimizing Applications with the Intel C++ and Fortran Compilers (accessed 26 April 07). `ftp://download.intel.com/software/products/compilers/techtopics/Compiler_Optimization_7_02.pdf`, 2004.

[13] R. Lysecky, G. Stitt, and F. Vahid. Warp processors. *ACM TODAES*, 11(3):659–681, 2006.

[14] R. J. Pankhurst. Operating systems: Program overlay techniques. *Commun. ACM*, 11(2):119–125, 1968.

[15] D. A. Patterson and J. L. Hennessy. *Computer organization & design: the hardware/software interface*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[16] I. Puaut and D. Hardy. Predictable paging in real-time systems: A compiler approach. In *Proc. ECRTS*, pages 169–178, Washington, DC, USA, 2007. IEEE Computer Society.

[17] I. Puaut and C. Pais. Scratchpad memories vs locked caches in hard real-time systems: a quantitative comparison. In *Proc. DATE*, pages 1484–1489, San Jose, CA, USA, 2007. EDA Consortium.

[18] P. Puschner and A. Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Syst.*, 18(2-3):115–128, 2000.

[19] R. Niemann and P. Marwedel. Hardware/software partitioning using integer programming. In *Proceedings of the European Design and Test Conference (ED & TC)*, pages 473–480, Paris, France, 1996. IEEE Computer Society Press (Los Alamitos, California).

[20] R. F. Rosin, G. Frieder, and J. Richard H. Eckhouse. An environment for research in microprogramming and emulation. *Commun. ACM*, 15(8):748–760, 1972.

[21] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET Centric Data Allocation to Scratchpad Memory. In *Proc. RTSS*, pages 223–232, Washington, DC, USA, 2005. IEEE Computer Society.

[22] D. W. Wall. Limits of Instruction-Level Parallelism. Technical Report WRL-93-6, DEC Western Research Laboratory, 1995.

[23] M. Ward and N. Audsley. Hardware compilation of sequential Ada. In *Proc. CASES*, pages 99–107, New York, NY, USA, 2001. ACM Press.

[24] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. In *Proc. Int. Conf. Quality Software*, Sep. 2005.

[25] J. Whitham. *Real-time Processor Architectures for Worst Case Execution Time Reduction*. PhD thesis, 2008.

[26] J. Whitham and N. Audsley. MCGREP - A Predictable Architecture for Embedded Real-time Systems. In *Proc. RTSS*, pages 13–24, 2006.

[27] J. Whitham and N. Audsley. A self-optimising simulator for a coarse-grained reconfigurable array. In *Proc. UK Embedded Forum*, pages 99–109. University of Newcastle, April 2007.

[28] J. Whitham and N. Audsley. Using trace scratchpads to reduce execution times in predictable real-time architectures. In *Proc. RTAS (to appear)*, 2008.

[29] Y. Xie and W. Wolf. Co-synthesis with custom asics. In *Proc. ASP-DAC*, pages 129–134, 2000.

[30] Xilinx. Virtex-4 Family Overview. Datasheet DS112, Xilinx Corporation, 2007.

[31] P. Yu and T. Mitra. Satisfying real-time constraints with custom instructions. In *Proc. CODES+ISSS*, pages 166–171, 2005.