Towards Parallel Programming Models for Predictability

Björn Lisper

School of Innovation, Design and Engineering, Mälardalen University
Box 883, S–721 23 Västerås, Sweden.
bjorn.lisper@mdh.se

Abstract

Future embedded systems for performance-demanding applications will be massively parallel. High performance tasks will be parallel programs, running on several cores, rather than single threads running on single cores. For hard real-time applications, WCETs for such tasks must be bounded. Low-level parallel programming models, based on concurrent threads, are notoriously hard to use due to their inherent nondeterminism. Therefore the parallel processing community has long considered high-level parallel programming models, which restrict the low-level models to regain determinism. In this position paper we argue that such parallel programming models are beneficial also for WCET analysis of parallel programs. We review some proposed models, and discuss their influence on timing predictability. In particular we identify data parallel programming as a suitable paradigm as it is deterministic and allows current methods for WCET analysis to be extended to parallel code. GPUs are increasingly used for high performance applications: we discuss a current GPU architecture, and we argue that it offers a parallel platform for compute-intensive applications for which it seems possible to construct precise timing models. Thus, a promising route for the future is to develop WCET analyses for data-parallel software running on GPUs.

1998 ACM Subject Classification C.3 Special-Purpose and Application-Based Systems

Keywords and phrases Real-Time System, WCET analysis, Parallel Program, Data Parallelism

Digital Object Identifier 10.4230/OASIcs.WCET.2012.48

1 Introduction

There is an ever growing need for advanced functionality in embedded systems. As hardware becomes more capable, new intelligent applications will emerge: some possible examples are real-time image processing and object recognition, multiple-sensor information fusion, and online spectral analysis for state-based maintenance. Some of these applications will have hard real-time constraints, and these constraints will then have to be verified. This requires a WCET analysis for the involved tasks.

High performance hardware is today invariably *parallel*, as this is the only way to meet the demands for computational power. A high performance task will also have to be a parallel program, thus utilizing the available computational resources. This raises the issue how to estimate the WCET of a parallel program, consisting of a number of cooperating threads synchronizing and exchanging data in some manner.

WCET analysis of parallel programs has hardly been studied up to now. Preliminary work on such a WCET analysis, for a simple language with parallel threads communicating through a shared memory and synchronizing with locks, is reported in [8]. A conclusion to be drawn from this work is that while it certainly is possible to define such an analysis, and prove its soundness, there will be issues to handle. One issue is that, since the functional

12th International Workshop on Worst-Case Execution Time Analysis (WCET 2012).
Editor: Tullio Vardanega; pp. 48–58

OpenAccess Series in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

semantics of the program now may be dependent on the timing, due to race conditions, it is no more possible to divide the analysis into a distinct program flow analysis, low-level analysis, and final calculation: the calculation has to be intertwined with the other phases to find the possible outcomes of race conditions, which may in turn affect the possible orders of writes, or acquirements of locks. Another issue is that the potentially many possible executions of such a program, especially in the presence of potential race conditions, may make the precision of the analysis sensitive to approximations. An overapproximated timing interval for a write, for instance, may introduce a "false" order of writes in the analysis that may in turn affect the estimated value of, say, a loop counter. This can in turn yield a very poor precision for the resulting WCET bound.

Thus, it is interesting to investigate programming models for parallel computers that avoid the nondeterminism of low-level, thread-based models, and that provide a more structured way to write parallel programs for high performance applications. Such models have been studied in the parallel computing community for the last 25 years, for two reasons: they provide easier and less error-prone programming, and they yield the possibility to attach abstract cost (performance) models to the parallel programming constructs. Both properties should be of interest also for embedded real-time software, in particular if the cost models can be sharpened to safe timing models.

In this position paper we argue that some of these parallel programming models provide a suitable parallel programming discipline for time-critical applications as well. In particular, we review the *data parallel* model, and we show how it restricts the patterns of parallelism in such a way that a WCET analysis can be done in a more or less traditional fashion if timing models are provided for the parallel constructs in this model. We identify for which constructs it may be problematic to find tight timing bounds – methods to find good bounds for them is a target for future research. We conclude by indicating how the data parallel constructs can be mapped to the CUDA Graphics Processing Unit (GPU) architecture.

The rest of the paper is organized as follows: in Section 2 we review the background: present and expected future development in embedded parallel computing, and the problems with unrestricted thread- and process based parallelism. Section 3 introduces the *Bulk Synchronous Parallel* model [21], which allows cost models that are valid over a wide range of parallel architectures, and in Section 4 we review the data parallel model and discuss timing models for the typical data parallel primitives. Section 5 introduces GPUs, and NVIDIA's CUDA model, and in Section 6 we discuss how the data parallel primitives can be mapped to CUDA. Section 7 provides a discussion of related work. In Section 8, finally, we conclude the paper and present future work.

2 Background

At this moment, multi-core processors are conquering the world. Current multi-cores have a traditional shared-memory architecture with a few cores sharing a common memory, connected via a shared bus. This architecture works reasonably well performance-wise for concurrent, independent tasks, or moderately parallel computing, but it does not scale since the shared bus and memory quickly become bottlenecks. As we go for higher performance, the number of cores will have to increase, distributed memory must be provided, and the bus must be replaced by a Network-on-Chip (NoC) that provides a wider communication channel. Tilera already offers general purpose many-cores with up to 100 cores, a distributed cache, and a mesh network[1].

[1] `http://www.tilera.com/products/processors`

Another strand of development is provided by GPUs. Originally designed to accelerate graphics, they are now rapidly turning into parallel general-purpose computational co-processors. We can expect them to be integrated on chip. Thus, a future processor architecture may have one part that is a conventional multi-core, with a few, relatively powerful processors, and a powerful computational co-processor with many relatively simple cores, connected through a NoC, and with local memories at different levels.

Now consider a time-critical, computationally intensive application, implemented by a parallel program, whose timing constraints must be verified. How can we obtain safe WCET estimates? Current practice in high performance computing it to write parallel software with explicit threads or processes, either using a shared memory, synchronizing through locks or semaphores (Pthreads [1], OpenMP [4]) or distributed memory, with synchronization and communication effectuated through message-passing (MPI [6]). Unfortunately such low-level parallel programming models are inherently nondeterministic, and race conditions create dependencies between timing and functional behaviour. This makes WCET analysis hard. Consider, for instance, the following example with parallel threads `T1` and `T2`, sharing the variable `n`:

```
T1: p; n = 100000000; halt
T2; q; n = 10; for i = 1 to n do r; halt
```

Here, there is a potential race condition for the assignment to `n`. If we know that $WCET(\texttt{p}) = 150$ and $BCET(\texttt{q}) = 200$, however, then `T2` will always "win" and the loop will always iterate ten times. But if an imprecise timing analysis yields a WCET bound for `p` that is greater than the BCET bound for `q`, then the analysis must assume that the loop can iterate 10^8 times. This example demonstrates that it is no more possible to perform a loop bounds analysis separately from the timing bounds calculation.

If we add synchronization through locks, then an imprecise analysis might even falsely detect deadlocks. If $BCET(\texttt{p}) > WCET(\texttt{q})$ below then no deadlock is possible, but again an imprecise timing analysis might fail to verify this and must then assume that a deadlock can occur.

```
T3: p; lock l; halt
T4; q; lock l; r; unlock l; halt
```

Also an imprecise program flow analysis might yield similar problems, like false deadlocks. Consider the two threads below, sharing a barrier `b`:

```
T5: m = expr1; for i = 1 to n do { a; barrier(b); c }
T6; m = expr2; for i = 1 to m do { x; barrier(b); y }
```

Assume that `expr1` amd `expr2` are such that they always evaluate to the same value. However, a value analysis may fail to detect this: a WCET analysis using the results from that value analysis for its loop bounds analysis will then have to assume that there is a possibility that one thread will attempt to execute its `barrier` statement one time more than the other one which then would cause it to deadlock. Thus, the analysis will have to assume that a deadlock might appear. (In fact, any loop bounds analysis that only detects *upper* loop bounds will not be strong enough to prove absence of deadlock for this example.)

These examples may be a bit contrived, but they are constructed to make a point. In reality, the kind of problems exemplified above may appear in much more subtle ways, in seemingly unproblematic code. The conclusion is that timing analysis of parallel software will be difficult as long as the programs are written on this level. To alleviate the problem, more disciplined parallel programming practices are needed that eliminate race conditions and the risk of deadlock.

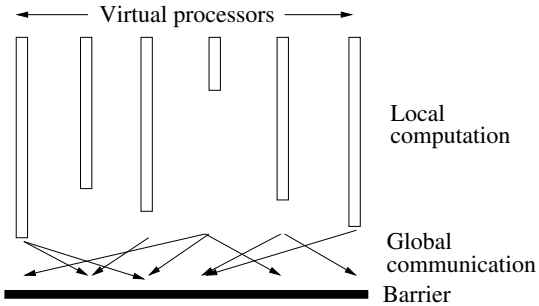

Figure 1 A BSP superstep (following [20]).

3 Bulk Synchronous Programming

Bulk Synchronous Programming (BSP) [20, 21] is a model for parallel computing allowing some performance analyzability over a wide range of target parallel architectures. The BSP model assumes a hardware model of *components* that are connected by a *router* (a network). The components can be processors or memories, and part of the router can connect single processors locally with memories: thus, the model covers both distributed memory architectures as well as shared (global) memory architectures, possibly with the memory structured into different memory banks.

BSP has a concept of *virtual processors*, where a physical processor in each computation step will execute the instructions of several virtual processors. In each computation step, the virtual processors execute independent tasks. To hide communication latencies it is assumed that there are considerably more virtual processors than actual processors, i.e., that the computation is massively parallel.

A BSP program executes in *supersteps*. In each superstep each component performs a task consisting of an initial phase of local computation steps, followed by a communication phase, followed by a final barrier synchronization. See Fig. 1.

The separation of computation, communication, and synchronization has some interesting consequences for the performance analyzability. For the computation phase, it is easy to estimate the execution time since the processors execute in an entirely local fashion. Similarly, the execution time for barrier synchronization should be easy to bound. Also note that the restriction to global barrier synchronization eliminates the risk of deadlock provided that all local supersteps terminate.

The communication phase is the hardest to analyze, since its performance is very dependent on the communication/access pattern, network topology, and distribution of processors and memories. The BSP model assumes that the router has a *permeability* g such that if each component (memory or processor) sends or receives at most h messages within a superstep, then the time for the communication phase is $O(gh)$. This assumption is easily violated in real parallel systems due to hotspots and similar. But assuming random (or hashed) distribution of data and computations such hotspots will be unlikely for a wide range of network topologies, and the permeability model can be applied to average case complexity calculations.

For WCET analysis it is clear that the separation into supersteps, with separate phases for computation, communication, and synchronization, should be beneficial. If the time for a superstep can be bounded, as well as the number of supersteps, then a coarse WCET estimate can be found by multiplying these bounds. The most serious remaining hurdle is

Elementwise application	`for all k in parallel do B[k] = f(A1[k],...,An[k])`
Get communication	`for all k in parallel do X[k] = Y[G[k]]`
Send communication	`for all k in parallel do X[G[k]] = Y[k]`
Replication	`for all k in parallel do X[k] = Y`
Masking	`for all k where B[k] in parallel do X[k] = ...k...`
Reduce	`reduce(op,X)` $=$ `X[0] op` $\cdots$ `op X[n` $-$ `1]`
Scan	`scan(op,X)` $=$ `[X[0], X[0] op X[1],...,X[0] op` $\cdots$ `op X[n` $-$ `1]]`

Figure 2 Data parallel operations.

the communication phase, which will require a much more detailed analysis for bounding the worst case tightly. Also, BSP does not preclude race conditions in this phase. Thus, parallel programming models that provide more predictable and deterministic communication primitives can be desirable from a timing analysis point of view.

4 The Data Parallel Programming Model

Data Parallel Programming is an instance of *Collection-Oriented Programming* [19], with roots in APL [10]. Collection-oriented programming emphasizes programming with homogenous data structures, such as arrays, lists, or sets. The core of the paradigm is a set of operations that work directly on such structures rather than on the elements one by one. This has two consequences: first, many loops can be replaced by such operations, rendering the programs more succinct and easier to understand, and second these primitives often have highly parallel implementations, with large numbers of small, independent computations. Thus, collection-oriented languages are interesting candidates for programming parallel hardware.

Data parallel languages take the collection-oriented approach one step further by associating each element in a data structure (often an array) with one virtual processor. The implementation will then map the virtual processors to physical processors, creating a partition of the data structure, and even effectuating a physical distribution in the case of a distributed memory machine.

The data parallel primitives can be classified in six groups: *elementwise application* applies a "scalar" function to each element in a data structure; *get communication* creates a new data structure by reading (pulling) each element from a source location in another data structure; *send communication* does likewise, but by *sending* (pushing) each element to a destination; *replication* creates a data structure whose elements are copies of a "scalar" value; *masking* will select part of a data structure for computation, using a boolean mask; *reduce*, and *scan*, finally, are generalized "sums" (or arrays of partial sums) where a binary, associative operator is successively applied to all elements in a data structure. See Fig. 2 for an informal definition, using a "`for all ... in parallel do`" syntax ranging over all virtual processors of the data structures.

If the elementwise applied functions are side effect-free, then *all data parallel operations except send communication are deterministic.* (For send communication there are write conflicts if `G[k]` $=$ `G[k']` for some `k` $\neq$ `k'`: however, if `G` specifies a permutation then this operation is also deterministic.) Thus, *a deterministic language that is extended with data parallel operations is still deterministic* (with the small proviso above).

It follows that a sequential language extended with data parallel primitives allows a conventional WCET analysis, with separate flow- and low-level analysis and calculation

phases, provided that the execution times for the data parallel operations can be bounded.

The timing bounds for the data parallel operations will depend strongly on how well the pattern of communication for an operation matches the communication capabilities of the network. This pattern in turn depends strongly on how the memory is organized, and how data is allocated in memory. Assume there are n virtual processors being executed on k physical processors. If communication can be done in unit time, then the execution time for elementwise application is $O(n/k)$ since the individual applications can be done in any order, on any processor. This bound will hold in particular if the data accessed by each virtual processor already resides in local memory for its physical processor. Under similar conditions reduce and scan can be done in $O(n/k + \log k)$ time, given that the network provides efficient support for tree-balanced summation. Masking will not add more than a small constant cost, and can be implemented for instance with predicated instructions. The remaining operations all concern general communication, for which a more detailed analysis of the actual access and communication patterns will be needed to bound the execution time reasonably well. However data parallel languages tend to offer specialized communication primitives, like array shifts, which provide efficient and predictable communication on a wide range of parallel hardware. For such primitives it will be easier to find tight timing bounds, and a parallel program that can use them instead of the general communication operations will likely be more timing predictable.

As for BSP, the strict division into communication, computation, and synchronization helps formulating the timing models. The data parallel paradigm has the additional advantage that it provides a set of distinct deterministic parallel operations: this helps timing analysis since timing bounds for these operations can be used directly in a conventional WCET calculation.

5 GPUs: the CUDA Model

GPUs have rapidly emerged as computational co-processors that provide high performance computing capabilities at low cost. A GPU is typically designed to run many similar, independent threads at high speed, using massive parallelism (up to hundreds of cores). It will have several levels of local memory, distinct from the host memory. Data is moved from host to local memory, and between different levels of local memory, by explicit operations. The host will issue jobs, consisting of a number of threads to execute to completion, to the GPU. The jobs can either block the host until completion, or allow that the host continues to execute concurrently.

Two rather similar application programming interfaces have emerged for GPUs: OpenCL, and NVIDIA's CUDA [14]. Both also provide a computation model. We now review CUDA and we also discuss briefly the current underlying execution model for NVIDIA GPUs.

The basic units of exeeution in CUDA are threads. The threads are grouped into *thread blocks*. A multithreaded CUDA program logically specifies a number of thread blocks: these are then scheduled in some unspecified, possibly parallel order on the GPU cores, where the execution of a thread block is done by executing its threads on a single core in some unspecified, possibly interleaved order. If there are no dependencies between the threads in a set of thread blocks, then the semantics will be independent of the order in which they are executed. This yields considerable flexibility in how thread blocks are executed, and the execution can easily adapt to the number of available cores in the GPU.

A thread within a block has a local unique ID within that block, and the block itself has a global unique ID: both can be accessed by the thread, and thus it can decide, e.g., which

part of an array that it will work on.

There is no synchronization available between thread blocks. Threads within a block can, however, synchronize through local barriers.

CUDA has a memory hierarchy, with memory at different levels of locality. Each thread has its own private memory. Each thread block has shared memory, with the same lifetime as the block, which is visible to all the threads in the block: variables residing in this memory are explicitly declared. There is also global, shared memory visible in all thread blocks. Local shared memory can be expected to be much faster than global shared memory. Furthermore, the CUDA model assumes that the GPU executes as co-processor to a host, and that their memories are distinct. Transfers between host and co-processor memory are done explicitly, through calls to the CUDA runtime system. A thread executing on the co-processor can only access memory that is located on that processor: data to be processed thus has to be copied from the host to the co-processor, and results must be copied back.

CUDA C extends C with *kernels*. These are C functions for which, when called, N instances are executed in parallel in N different CUDA threads. A special call to the kernel will issue the execution of the threads on the GPU.

A CUDA C program will alternatingly execute in sequential and parallel phases. In a sequential phase, the C program is executing as usual on the host. A call to a kernel invokes a parallel phase where the kernel is executed in parallel, in a number of thread blocks, on the co-processor (the GPU). When the kernel call returns, the sequential execution will resume.

Under some circumstances the co-processor can execute concurrently with the host. Control is then returned to the host before the call has completed. This includes asynchronous kernel calls, and certain memory transfers.

NVIDIA's underlying hardware architecture is called SIMT (Single-Instruction, Multiple-Thread), and it can be seen as a hybrid of SIMD and MIMD with separate threads but strong coherence locally between their execution. Threads are executed on a scalable array of multithreaded *Streaming Multiprocessors* (SMs). The execution of threads is organized in groups of 32 threads called *warps*. A number of warps will make up a thread block. All threads in a warp will execute on the same SM: they will start at the same address, but they have their own program counters. Threads in a warp can therefore take different execution paths through the code. However, different threads in a warp can execute simultaneously only if they execute the same instruction: thus, diverging threads will force the warp to execute the instructions on each path separately, disabling threads that are not on that path. This can affect the performance very adversely. Instructions can be predicated, though.

The warps are scheduled by a *warp scheduler*. This scheduler will always pick a warp with an instruction ready to execute, if available. Since there typically will be considerably more warps than cores to execute them on, this will help mask the memory latencies as it allows memory fetches issued by instructions to complete in parallel with instructions for other warps being executed.

The current SM cores have a fairly simple architecture: instructions are pipelined, but they are executed in order and there is no branch prediction or speculative execution. Each SM has a set of registers, and local memory partitioned among the thread blocks: on some GPU models this is a partitioned cache, whereas for others it is explicitly addressed. A GPU also has global memory that is shared among the SMs.

The conclusion is that overall, the NVIDIA architecture with the CUDA model should be quite amenable to timing analysis. The processors are simple. Memory access times vary strongly with the degree of locality, but for the most part it is explicit which part of memory is being accessed. GPU models with local caches will need a cache analysis,

though. Although threads are executed concurrently, it is done in a very controlled fashion. Also the interaction with the host is very explicit, with calls for submitting kernels and transferring memory contents. Overlaps in execution between GPU and host should be possible to estimate. Finally, since the GPU executes requests in sequence there will not be the same competition for shared resources within the GPU, between unrelated activities, as in a bus-based multi-core processor.

The largest sources of uncertainty seem to be (1) the fact that a warp can only execute threads executing the same instruction in parallel, and (2) uncertainty about exactly how the warps are scheduled. (1) will be alleviated if the executed kernels have a predictable, indata-independent program flow: if not, then an analysis may have to assume that all 32 instances of the kernel in a warp will diverge, and thus must have their respective instructions sequentially executed in each step. As for (2), uncertainties about warp scheduling means that it will be harder to estimate the overlaps between warp execution and memory fetches.

6 Mapping Data Parallel Primitives to GPUs

It should be evident from the preceding discussion that data parallel programming and GPUs is a good match. GPUs are designed to execute a large number of similar threads, executing in the same fashion with little or no synchronization, in parallel. The data parallel operations described in Section 4 can be implemented by such threads. For elementwise applied operations this is obvious. Parallel read operations, and replication, can also be implemented by threads reading data to local shared memory, and parallel writes will copy data in parallel from local memories to global shared memory. Masking can be implemented efficiently by predicated execution of instructions. A compiler can coalesce sequences of such data parallel operations into kernel code implementing them on the GPU: these kernels will execute efficiently and predictably since all threads will execute the same path. Operations like reduce and scan, finally, also have efficient GPU implementations [18]. Thus, data parallel languages implemented on GPUs should provide an interesting alternative for implementing high performance applications with stringent timing constraints.

7 Related Work

Most work regarding WCET analysis for parallel systems concern single-threaded programs running on single cores in multi-cores. Here, the challenge is to find tight WCET bounds when different activities can compete for shared resources like buses and memories [2, 3]. A common assumption is that the system bus uses TDMA to provide predicable bandwidth for the different cores [17]. Various cache analyses have also been considered for multi-cores [9, 11, 12].

Current multi-core architectures have poor timing predicability due to the presence of shared resources under loose control. Principles for designing timing-predictable multi-core hardware and software have been devised [22]. A multi-core architecture providing timing predictability for hard real-time tasks was suggested in [15].

WCET analysis of parallel programs, running on parallel hardware, has not been much studied. A case study using timed automata was reported in [7]. Another case study is found in [16]. In [8], a general WCET analysis for parallel programs with threads, shared memory, and locks is devised and its correctness is proved for the restricted class of programs that do not use locks. This analysis integrates flow analysis and WCET calculation using abstract execution in a manner similar to [5].

In [13], a simple cost model for a GPU program was estimated from measurements and used in a subsequent WCET calculation. No systematic WCET analysis was defined, though.

8 Conclusions and Further Research

WCET analysis of parallel software can easily become very difficult. Sources of unpredictability are shared hardware resources, providing channels for different activities to affect each others' timing, and low-level concurrent programming models that are inherently nondeterministic.

We discuss the use of restricted parallel programming models to make parallel software more amenable to timing analysis, with a focus on models for performance-demanding real-time applications. Such models have been proposed in the past, as means to make parallel programming easier and to allow general performance models. We review BSP and data parallel programming, and we conclude that especially the latter provides a promising model for writing timing-predictable parallel software.

We furthermore review a common current GPU architecture and computation model, and conclude that it seems to provide a reasonably timing-predictable platform for high performance computing applications. The combination of data parallel programming and GPU execution seems especially interesting, since the data parallel primitives can be translated into kernels that will make timing-predictable use of the streaming multiprocessors in the GPUs.

A possible development of future hardware architecture is that we will see heterogenous processors with a few general purpose multi-cores, and an on-chip massively parallel co-processor, building on GPU technology, for computationally demanding applications. Already now Broadcom offers the BCM2835 System-on-Chip with an ARM11 processor and a Videocore 4 GPU. Such future processors will surely find their way into various embedded real-time applications. This means that we will need timing analysis for these systems. In this paper, we have laid out a path how to get there. However, other directions are also possible: we may also see future many-core processors along the lines of the Tilera architecture, with a large number of general purpose cores equipped with distributed memory and connected by a NoC. The data parallel paradigm also provides good support for such architectures due to its deterministic primitives, and their separation into primitives for computation and communication. It seems likely that good timing models should be possible to develop for these primitives also when implemented on such architectures.

Future work will include designing suitable timing models and low-level analyses for future parallel architectures, and design WCET analyses that that can use these to provide tight and safe WCET bounds.

Acknowledgment

This research was supported by the Swedish Foundation for Strategic Research through grant IIS11-0060 (Ralf 3 – Software for Embedded High Performance Architecture).

References

1 David R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.

2 Sudipta Chattopadhyay, C.-L. Kee, Abhik Roychoudhury, Timon Kelter, Peter Marwedel, and Heiko Falk. A unified WCET analysis framework for multi-core platforms. In

18th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS'12), Beijing, China, April 2012.

3 Sudipta Chattopadhyay, Abhik Roychoudhury, and Tulika Mitra. Modeling shared cache and bus in multi-cores for timing analysis. In Todor Stefanov, editor, *Proc. 13th International Workshop on Software and Compilers for Embedded Systems (SCOPES'10)*, pages 6:1–6:10, St. Goar, Germany, June 2010. ACM.

4 Leonardo Dagum and Ramesh Menon. OpenMP: An industry-standard API for shared-memory programming. *Computing in Science and Engineering*, 5(1):46–55, 1998.

5 Andreas Ermedahl, Jan Gustafsson, and Björn Lisper. Deriving WCET bounds by abstract execution. In Chris Healy, editor, *Proc. 11th International Workshop on Worst-Case Execution Time Analysis (WCET'2011)*, Porto, Portugal, July 2011.

6 Message Passing Interface Forum. MPI: A message passing interface standard. Technical report, University of Tennessee, Knoxville, Tennessee, May 1994.

7 Andreas Gustavsson, Andreas Ermedahl, Björn Lisper, and Paul Pettersson. Towards WCET analysis of multicore architectures using UPPAAL. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 103–113, Brussels, Belgium, July 2010. OCG.

8 Andreas Gustavsson, Jan Gustafsson, and Björn Lisper. Toward static timing analysis of parallel software. Accepted for publication in 12th International Workshop on Worst-Case Execution Time Analysis (WCET'2012).

9 Damien Hardy, Thomas Piquet, and Isabelle Puaut. Using bypass to tighten WCET estimates for multi-core processors with shared instruction caches. In *Proc. 30th IEEE Real-Time Systems Symposium (RTSS'09)*, pages 68–77, 2009.

10 K. E. Iverson. *A Programming Language*. Wiley, London, 1962.

11 Y. Li, Vivy Suhendra, Y. Liang, Tulika Mitra, and Abhik Roychoudhury. Timing analysis of concurrent programs running on shared cache multi-cores. In *Proc. 30th IEEE Real-Time Systems Symposium (RTSS'09)*, Washington, D.C., USA, December 2009.

12 Mingsong Lv, Nan Guan, Wang Yi, and Ge Yu. Combining abstract interpretation with model checking for timing analysis of multicore software. In Scott Brandt, editor, *Proc. 31th IEEE Real-Time Systems Symposium (RTSS'10)*, pages 339–349, San Diego, CA, December 2010. IEEE.

13 Rahul Mangharam and Aminreza Abrahimi Saba. Anytime algorithms for GPU architectures. In Luis Almeida, editor, *Proc. 32nd IEEE Real-Time Systems Symposium (RTSS'11)*, pages 47–56, Vienna, Austria, December 2011. IEEE Computer Society.

14 NVIDIA CUDA C programming guide, November 2011.

15 Marco Paolieri, Eduardo Quiñones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *Proc. 36th International Symposium on Computer Architecture (ISCA 2009)*, pages 57–68, 2009.

16 Christine Rochange, Armelle Bonenfant, Pascal Sainrat, Mike Gerdes, Julian Wolf, Theo Ungerer, Zlatko Petrov, and Frantisek Mikulu. WCET analysis of a parallel 3D multigrid solver executed on the MERASA multi-core. In Björn Lisper, editor, *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'2010)*, pages 90–100, Brussels, Belgium, July 2010. OCG.

17 Jakob Rosen, Alexandru Andrei, Petru Eles, and Zebo Peng. Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip. In *Proc. 28th IEEE Real-Time Systems Symposium (RTSS'07)*, pages 49–60, Washington, DC, USA, 2007. IEEE Computer Society.

18 Shubhabrata Sengupta, Mark Harris, Yao Zhang, and John D. Owens. Scan primitives for GPU computing. In *Proc. 22nd ACM SIGGRAPH/EUROGRAPHICS symposium on*

Graphics hardware, GH '07, pages 97–106, Aire-la-Ville, Switzerland, 2007. Eurographics Association.

19 Jay M. Sipelstein and Guy E. Blelloch. Collection-oriented languages. *Proc. IEEE*, 79(4):504–523, April 1991.

20 David B. Skillicorn, Jonathan M. D. Hill, and W. F. McColl. Questions and answers about BSP. *Scientific Programming*, 6:249–274, 1997.

21 Leslie G. Valiant. A bridging model for parallel computation. *Comm. ACM*, 33(8):103–111, August 1990.

22 Reinhard Wilhelm. The PROMPT design principles for predictable multi-core architectures. In *Proc. 12th International Workshop on Software and Compilers for Embedded Systems (SCOPES'09)*, pages 31–32, Nice, France, April 2009. ACM.