Advances in c-based parallel design of MP-SOCs

MARTTI FORSELL
Platform Architectures Team
VTT Technical Research Center of Finland
Box 1100, FI-90571 Oulu, Finland

Abstract: As the main stream of system-on-a-chip (SOC) architectures is gradually switching from single processor to multiprocessor (MP) constellations, availability of easy-to-use/migrate parallel design methodologies are becoming more and more important. C-based design methodologies provide potentially easy migration path in SOC design, but they have traditionally lacked general purpose and easy-to-use tools for exploiting parallelism. However, recent advances in c-based design of MP-SOCs, like introduction of the e-language—a simple parallel extension of c for a class of emulated shared memory MP-SOCs—and related design methodology promise to overcome these problems. In this paper we describe latest advances in e-based design, including an initial implementation of e for concurrent memory access-ready architectures, fast mode providing a significant boost in parallel construct performance for simple e-programs, and support for active memory operations that drops the lower bound of the execution time of certain logarithmic algorithms to the constant execution time class.

Key-words: Parallel computing, parallel languages, optimization, PRAM model, CRCW, active memory

1 Introduction

Systems-on-a-chip (SOC) are among the key components of current and future electronic devices. Their variability and small size benefit especially feature driven smart phone, communicator, personal digital assistant, and mobile computer markets in which programmability, low power usage, and fast time-to-market are crucial in addition to adequate performance. These requirements together with demand for more performance and changes in forthcoming silicon technology limiting the practical maximum clock rate of a chip and making global point-to-point connections infeasible [ITRS05] are pushing the main stream of SOC architectures gradually away from single processor constellations towards something that can be called multicore or multiprocessor (MP) constellations [Taylor02, Sankaralingam03]. As a result of this, current sequential computing-based design methodologies [Balarin97, Chang99] need to be replaced with easy-to-use/migrate methodologies allowing efficient exploitation of parallel functionality on parallel MP-SOC hardware.

C-based design methodologies provide easy migration path in SOC design, but they have traditionally lacked general purpose and easy-to-use tools for exploiting parallelism even if special hardware design oriented languages e.g. SystemC or parallel libraries e.g. MPI or OpenMP are counted. However, recent advances in c-based design of MP-SOCs, like introduction of e-language—a simple parallel extension of c for a class of emulated shared memory MP-SOC architectures [Forsell04]—and related design methodology [Forsell05d] promise to overcome these problems by supporting high-level c-like access to fine-grained *thread-level parallelism* (TLP) and synchronous shared memory abstraction making program and data partitioning and synchronization simple. Unfortunately, the initial implementation of the e-language introduced quite high parallel construct execution time overheads and was limited to *exclusive read exclusive write* (EREW) memory access model only [Forsell04b]. In this paper we describe latest advances in e-based design, including an implementation of e for *concurrent read concurrent write* (CRCW) architectures, special fast mode providing a significant boost in parallel construct performance for simple e-programs containing a limited number of constructs and barrier synchronizations, and support for active memory operations that drops the lower bound of the execution time of certain logarithmic algorithms to the constant execution time class.

The rest of this article is organized so that in section 2 the e-language and a class of target MP-SOC architectures are described. In section 3 we list advances in e-based parallel design methodology for the target MP-SOCs and give examples of applying them to sim-

ple parallel problems. A brief performance and code size evaluation of the described techniques is given in section 4. Finally, in section 5 we give our conclusions.

2 E-language and target MP-SOC architectures

In order to support both easy-to-use and efficient design of functionality for MP-SOCs, a c-like high-level parallel programming language e and corresponding class of advanced MP-SOC architectures have been introduced [Forsell04, Forsell02].

2.1 E-language

The e-language [Forsell04] is an experimental TLP programming language created especially for emulated shared memory MP-SOCs providing general purpose functionality, but it can be used also for multichip synchronous shared memory architectures conforming the IPSM framework [Forsell97]. The syntax of e-language is an extension of the syntax of the familiar c-language. E-language supports parallely recursive and synchronous *multiple instruction stream multiple data stream* (MIMD) programming and it is intended to work with various *parallel random access machine* (PRAM) models [Keller01], but current implementations include the EREW PRAM and ideal CRCW PRAM models only. (The EREW PRAM model allows a single reference per a shared memory location only while the CRCW PRAM allows arbitrary concurrent read concurrent write access to any location.)

Variables in the e-language can be shared among a group of threads or they can be private to a thread. Using shared variables as modal parameters or result value of a function is not supported in the current implementation of e. If an actual parameter is a shared variable, private copies of value or reference will be used in the function execution.

In order to support high-level TLP expressions with the MIMD paradigm, threads form hierarchical groups that are automatically numbered from 0 to the number of threads-1 as new groups are created (alternatively, a programmer can use static thread numbering). In the beginning of a program there exists a single group containing all threads. A group can be divided into subgroups so that in each thread of the group is assigned into one of the subgroups. A subgroup may be split into further subgroups, but the existence of each level of subgroups ends as control returns back to the corresponding parent group. As a subgroup is created, dynamic thread identification variables are updated to reflect the new situation and as the subgroups join back to the parent group in the end of the statement the old values of these variables are restored.

Emulated shared memory architectures and the IPSM framework guarantee synchronous execution of instructions at machine instruction level. In the e-language synchronicity and/or automatic subgroup creation through control structures having private enter/exit conditions can be maintained with special versions of control structures supporting automatic synchronization at the end of the structure, supporting automatic subgroup creation, or both automatic synchronization and subgroup creation (see Figure 1). There are two control modes in e: In the first control mode, called synchronous area, all threads belonging to the current group execute the same portion of code synchronously. Asynchronous control structures, i.e. those with private enter/exit conditions but without synchronization, can be used only at the leaf level of group hierarchy. Entering to an asynchronous area, the second control mode, happens by using an asynchronous control structure and returning back to the synchronous area happens by an explicit group-wide barrier synchronization assuming all threads of the group will reach the barrier.

Structure	Calling Area	Create subgroups	Synch at the end
if (c) s;	Both	-	no
if (c) s1; else s2;	Both	-	no
while (c) s;	Both	-	no
do s while (c);	Both	-	no
for (s1;s2;s3) s;	Both	-	no
if_ (c,s);	Both	-	yes
if_else_ (c,s1,s2);	Both	-	yes
while_ (c,s);	Both	-	yes
do_while_ (s,c);	Both	-	yes
for_ (s1,s2,s3,s);	Both	-	yes
_if (c,s);	Synch	1	no
_if_else (c,s1,s2);	Synch	2	no
_while (c,s);	Synch	1	no
_do_while (s,c);	Synch	1	no
_for (s1,s2,s3,s);	Synch	1	no
_if_ (c,s);	Synch	1	yes
_if_else_ (c,s1,s2);	Synch	2	yes
while (c,s);	Synch	1	yes
_do_while_ (s,c);	Synch	1	yes
for (s1,s2,s3,s);	Synch	1	yes

Figure 1. The control structures of E-language.

A typical design flow for implementing required functionality on a MP-SOC with e-language consists of eight parts (see Figure 2).

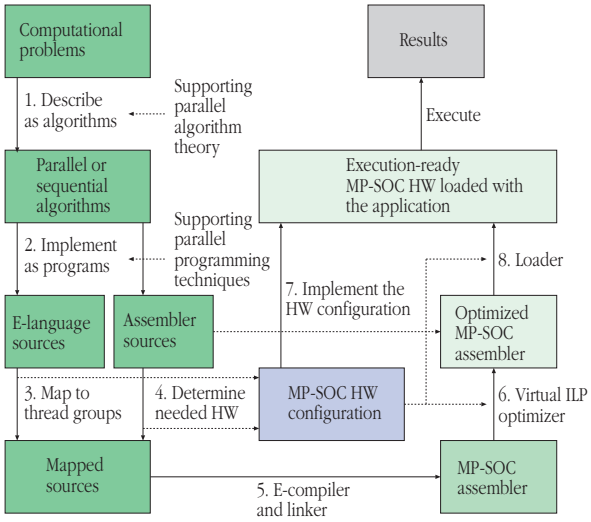

Figure 2. The development flow for the e-language on a target MP-SOC architecture.

1. Describe computational problems as algorithms. In this part a designer describes computational problems forming the application as parallel or sequential algorithms. If a problem is solved with one or more parallel algorithms, it needs to be divided into subtasks executable with parallel threads. At this point one may also want to make the degree of subtasks adjustable so that the same algorithm can be used with other MP-SOC configurations, e.g. having different number of threads. All this happens according to the theory of parallel algorithms and with the help of available algorithm libraries [Jaja92, Keller01].

2. Implement algorithms as programs. In this part a designer implements the obtained parallel and sequential algorithms as parallel and sequential e-language and assembler programs. It includes locating data requiring interaction between multiple subtasks into the shared memory, managing the degree of parallel access, taking care of synchronization of subtasks over task-private control structures, and managing parallel I/O. There exists a rich set of parallel programming techniques supporting this kind of activity [Keller01].

3. Map tasks to thread groups. In this part a designer maps tasks to thread groups according to performance requirements, and adds synchronization primitives and program intercommunication data areas where necessary. Threads can be selected from the set of available threads in an arbitrary way. In the PRAM model threads do not interfere with each other unless they are ordered to do so via the shared memory. The result of this part should be a single parallel program consisting of one or more files written in e and/or assembler.

4. Determine the needed hardware. In this part a designer determines the needed hardware including the number of threads, the number and type of *functional units* (FU) per processor, the amount of shared and private memory needed, unless the underlying MP-SOC is fixed by the used platform. The appropriate number of FUs and threads for each task can be determined with the help of parallel algorithm theory, test execution and simple calculations.

5. Compile and link. In this part the obtained program files are compiled with e and assembler compiler and linked together. This part may also include applying required standard c-based optimizations to the source files.

6. Optimize virtual instruction-level parallelism. In this part the compiler output is further optimized for the instruction-level parallel (ILP) computing model of the MP-SOC. This happens by applying the the virtual ILP optimization algorithm [Forsell03]. Inclusion of scheduled assembler is also possible at this point to support hand optimization.

7. Implement the hardware configuration. In this part the MP-SOC hardware is implemented according to the requirements obtained from part 4 unless the hardware is already fixed by the platform in use.

8. Load the code to the MP-SOC. In this part the optimized program is loaded to the target MP-SOC. The program loader takes care of placing instructions to instruction memory and variables to appropriate data memory subspaces. In addition it binds e-language primitives, e.g. synchronization and group creation and maintenance, to corresponding run time library routines.

An experimental parallel computing learning set, ParLe, for configurable shared memory MP-SOCs and corresponding ideal PRAM has been implemented [Forsell05c]. The learning set consists of an experimental optimizing compiler for e and assembler, linker, loader, simulator with a graphical user interface and statistical tools, and sample e/assembler code. Using the set, a student/designer can easily write simple parallel programs, compile and load them into a configurable MP-SOC platform, debug/execute them, gather statistics and explore the performance, utilization, and

obtain gate count estimations with different architectural parameters. The learning set runs on Mac OS X systems, supports currently only EREW MP-SOCs and is available for non-profit educational purposes.

2.2 Target MP-SOC architectures

In this section we will define a class of emulated shared memory MP-SOC architectures providing a strong model of computing capable for synchronous concurrent memory access, e.g. by realizing CRCW PRAM with a fixed number of physical threads, denoted here as T. An MP-SOC belonging to the class is composed of P T_p-threaded multithreaded processors, a number of memory modules, and high-capacity interconnection network (see Figure 3). It is evident that at least Eclipse, a general purpose MP-SOC architecture being developed at VTT [Forsell02, Forsell05a], belongs to the defined class.

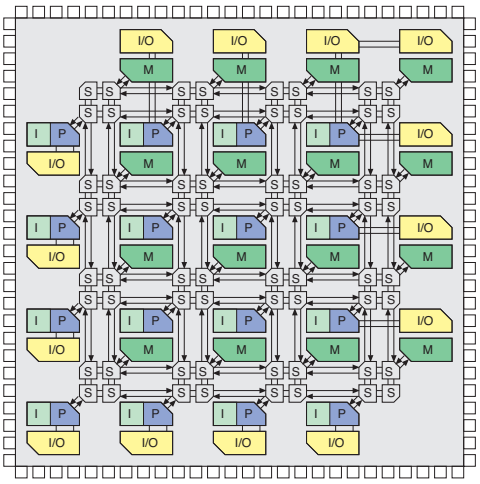

Figure 3. High-level block diagram of a sparse mesh-based MP-SOC architecture. (P=processor core, M=embedded memory/data memory module, rni=resource network interface, I=instruction memory module, I/O=I/O device and S=switch.)

The strong model of computing provides an uniform shared data memory with single step memory access latency and machine instruction level synchronous execution. All the threads are executing the same program, but may branch independently within the program. The model allows arbitrary concurrent reads and writes to memory locations. For a concurrent read all threads participating the access give the same results. In the case of a concurrent write, the data of an arbitrary thread participating the write will be written to the target location. The programming model of e and architecture defined computing model are linked together so that proceeding a full cycle in the pipeline corresponds typically to a single PRAM step. During a step, each thread of each processor of the MP-SOC executes, one by one, an instruction, which may include at most one shared memory reference.

In order to implement the synchronous shared memory abstraction on a MP-SOC, the communication network needs to connect processors to distributed memory modules so that high throughput and acceptable latency can be achieved for arbitrary communication patterns. We assume also that architectures support a small number of multioperations and arbitrary ordered multiprefix operations that can be used e.g. to add data words provided by a group of threads in constant time within a memory location, and to implement arbitrary and flexible barrier synchronizations between threads [Forsell05b]. Active memory realization adds an active memory unit consisting of a simple ALU and fetcher to each memory module.

3 Advances in e-language design

Latest advances in e-language design include an implementation of e for CRCW-ready architectures, special fast mode, and support for active memory operations.

3.1 Support for concurrent read and concurrent write

We have created an initial e implementation for CRCW-ready architectures by rewriting e-language construct support code for the CRCW model: We transformed complex limited active memory based concurrent accesses on a top of the EREW model to real CRCW accesses, reduced the number of implicit support variables by eliminating the internally used subgroup concept, modified the thread group frames used in subgroup creation accordingly, and rewrote the run time library eRunLib to reflect better the CRCW functionality.

Consider adding a number to elements of an array in parallel. In the case of the EREW model one must make thread-wise copies of the number before adding while in the case of the CRCW model one may just add the number to each element in parallel (see Figure 4).

```
int a_;            // A shared variable
int b_[size];      // A shared array of integers
int tmp_[size]; // Thread-wise copies of a_

// EREW version:
int i;
// Spread a_ to tmp_ with a logarithmic algorithm
if_ ( _thread_id==0 , tmp_[0]=a_; );
for (i=1; i<_number_of_threads; i<<=1)
    if_ ( _thread_id-i>=0 ,
        tmp_[_thread_id]=tmp_[_thread_id-i]; );
b_[_thread_id]+=tmp_[_thread_id]

// CRCW version:
b_[_thread_id]+=a_;
```

Figure 4. Add a_ to elements of b_ in parallel.

3.2 Support for active memory operations

Active memory operations can be used to boost the performance of MP-SOCs so that certain algorithms that have a logarithmic lower bound of execution time will execute in constant time [Forsell05c]. The underlying class of architecture supports now active memory operations for all memory locations, not just a small part of it. We added four primitives to e-language to support this kind of active memory operations:

prefix(p,M,m,c)	Perform a two instruction arbitrary multiprefix operation *OP* for components *c* in memory location *m*. The results are returned in *p*.
fast_prefix(p,OP,m,c)	Perform a single instruction multiprefix operation *OP* for at most *O*(square root *T*) components *c* in memory location *m*. The results are returned in *p*.
multi(OP,m,c)	Perform a two instruction arbitrary multioperation *OP* for components *c* in memory location *m*.
fast_multi(OP,m,c)	Perform a single instruction multioperation *OP* for at most *O*(square root *T*) components *c* in memory location *m*.

Consider calculating the sum of elements of an array in parallel. In the case of the EREW model one must use a logarithmic algorithm to obtain the sum while in the case of the active memory functionality one may just call multioperation primitive *multi* utilizing the BMADD and EMADD instructions implementing summation in constant time (see Figure 5).

```
int sum_;        // A shared variable
int a_[size];    // A shared array of integers

// EREW version—logarithmic algorithm for sum
for_ ( i=1 , i<_number_of_threads , i<<=1 ,
    if (_thread_id-i>=0)
        a_[_thread_id] += a_[_thread_id-i]; );
sum_=a_[_number_of_threads-1]

// Active memory version
//—just call the constant time sum primitive:
multi(MADD,&sum_,a_[_thread_id]);
```

Figure 5. Sum the elements of a_ into sum_ in parallel.

3.3 Fast mode

The initial e-language implementation introduced a quite high execution time overheads to parallel primitives [Forsell04b]. The switch from EREW to CRCW has dropped those overheads, but still they may limit the applicability of e-language in certain high performance requirements applications. In order to further cut those overheads we have implemented a special fast mode for simple non-structural e-programs that provides higher performance but limited feature set. The fast mode restrictions include:

- Overlapping execution of e-language constructs employing subgroup creation or barrier synchronization by multiple thread groups is not allowed
- Shared variables local to functions are not supported
- Subgroup specific thread numbering will not be passed automatically across the subroutine borders
- The number of simultaneously active barriers is limited by the underlying MP-SOC architecture

The fast mode can be applied to e-language programs simply with a compiler option -fast. To extend the applicability of the fast mode a programmer may chose explicit numbering of fast mode primitives. Finally, a programmer can explicitly access some fast mode specific features, like fast barrier synchronization, from ordinary e-programs.

Consider a portion of code from constant time sorting algorithm containing a rank function call in which

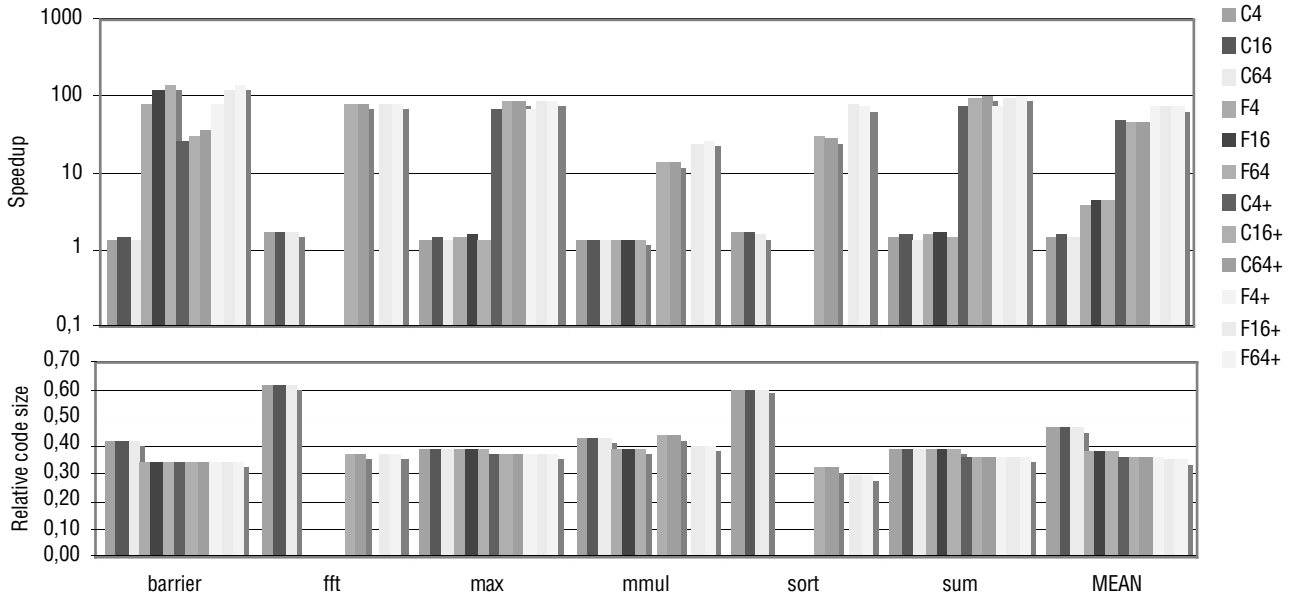

Figure 8. Relative performance (top) and code size (bottom) of MP-SOC configurations with respect to corresponding E4, E16, and E64 EREW configurations.

the called function uses subgroup relative thread numbering. It can be transformed to fast mode compliant code by inlining the function or by passing the thread numbers as parameters (see Figure 6).

```
int src_[N]; // Array to be sorted
int tgt_[N]; // Rank for each element

Flexible mode code:
void rank() // Parallel rank function
{   int i = _thread_id >> logn;
    int j = _thread_id & (N-1);
    fast_multi(MADD,&tgt_[j],src_[i]<src_[j]); }
// Calling code
if_ ( _thread_id<N2 , rank() );
if_ ( _thread_id<N ,
    src_[-tgt_[_thread_id]]=src_[_thread_id]; );

// Fast mode code:
int i = _thread_id >> logn;
int j = _thread_id & (N-1);
if_ ( _thread_id<N2 ,
    fast_multi(MADD,&tgt_[j],src_[i]<src_[j]););
if_ ( _thread_id<N ,
    src_[-target_[_thread_id]]=src_[_thread_id]; );
```

Figure 6. Sort src_ in parallel.

4 Evaluation

In order to illustrate the improvements achievable with the new e features on realistic MP-SOCs we mapped six parallel programs (barrier synchronization, fast fourier transform, maximum find, matrix multiplication, integer sort, and sum of an array) to thread groups, compiled, optimized (-O2 -ilp) and loaded them to 15 MP-SOC configurations having 4, 16 and 64 five FU 512-threaded processors (see Figure 7), and executed them with the IPSMSim simulator [Forsell05c].

Configuration	*P*	Model	Fast mode	Active memory
E4	4	EREW	no	no
E16	16	EREW	no	no
E64	64	EREW	no	no
C4	4	CRCW	no	no
C16	16	CRCW	no	no
C64	64	CRCW	no	no
F4	4	CRCW	yes	no
F16	16	CRCW	yes	no
F64	64	CRCW	yes	no
C4+	4	CRCW	no	yes
C16+	16	CRCW	no	yes
C64+	64	CRCW	no	yes
F4+	4	CRCW	yes	yes
F16+	16	CRCW	yes	yes
F64+	64	CRCW	yes	yes

Figure 7. MP-SOC configurations used in experiments.

The executable sizes including initialization and run-time libraries showing notable 55-65% decrease in instruction memory allocation as well as the execution

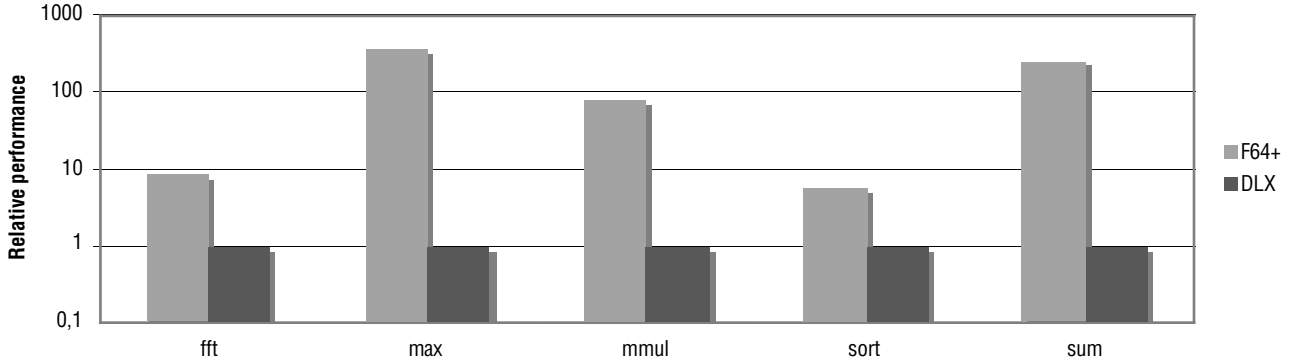

Figure 9. Performance of an F64+ MP-SOC utilizing fast mode e with respect to that of a single DLX processor system utilizing sequential c.

times excluding initialization of data showing speedup from 50% to a whopping 7200% are illustrated in Figure 8.

For comparison purposes, we implemented sequential c-versions of the programs, compiled them with the c-compiler (being also the main part of the e-compiler) with the same optimization settings, and executed the obtained binaries on the five stage pipelined DLX processor [Hennessy90] attached to an ideal memory system. The obtained execution times of the sequential versions are compared to those of fastest parallel versions (both fast mode and active memory support) on the 64 processor configuration in Figure 9. The results are somewhat expected indicating that the performance of fast mode e is well comparable to that provided by sequential c. The speedups for fft and sort are lower than for the rest of the programs since they are based on non-work optimal algorithms dropping the theoretical speedup from $O(N)$ to $O(\log N)$.

5 Conclusions

We have described recent advances in e-language development, including an initial implementation of e for CRCW-ready architectures, fast mode providing significant boost in parallel construct performance for simple e-programs, and support for active memory operations. According to our evaluation speedups of 1.5, 4.3, 47 and 73 are achieved with respect to the plain EREW versions for a set of simple parallel benchmarks by employing initial CRCW-ready implementation of e, fast mode, active memory support, and both fast mode and active memory support, respectively. It should be noted that part of the speedup is due to faster constant time multioperation and CRCW algorithms not only the more efficient implementation of e. The code size drops to 46% for the CRCW-ready implementation and roughly to 35% for the rest of the techniques. We compared the performance of fast mode active memory supported e to that of sequential c on a standard 5-stage pipelined processor. The achieved speedups range from 5.5 to a whopping 346. Thus, we conclude that these new advances in the e-language open up a spectrum of new possibilities to apply easy-to-migrate parallel c-like design to products employing MP-SOCs also when solutions are performance and code size critical.

Our plans for future research in this area include further improving the architecture, e-language and tool set with better syntax, more natural support for all kinds of parallelism, and flexibility/configurability for certain special cases where the computation should be arranged better to obtain higher performance.

Acknowledgements:

This work was supported by the grant 107177 of the Academy of Finland.

References:

[Balarin97] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Sangiovanni-Vincentelli, K. Suzuki, Hardware-Software Co-Design of Embedded Systems—The POLIS Approach, Kluwer Academic Publishers, Boston, 1997.

[Chang99] H. Chang, L. Cooke, M. Hunt, G. Martin, A. McNelly, and L. Todd, Surviving the SOC Revolution—A Guide to Platform-Based Design, Kluwer Academic Press, Boston, 1999.

[Forsell97] M. Forsell, Implementation of Instruction-Level and Thread-Level Parallelism in Computers, Dissertations 2, Department of Computer Science, University of Joensuu, Finland, 1997.

[Forsell02] M. Forsell, A Scalable High-Performance Computing Solution for Network on Chips, IEEE Micro 22, 5 (September-October 2002), 46-55.

[Forsell03] M. Forsell, Using Parallel Slackness for Extracting ILP from Sequential Threads, In the Proceedings of the SSGRR-2003s, July 28 - August 3, 2003, L'Aquila, Italy.

[Forsell04] M. Forsell, E—A Language for Thread-Level Parallel Programming on Synchronous Shared Memory NOCs, WSEAS Transactions on Computers 3, 3 (July 2004), 807-812.

[Forsell04b] M.Forsell, Compiling Thread-Level Parallel Programs with a C-Compiler, In the Proceedings of the IV Jornadas sobre Programacion y Lenguajes (PROLE'04), November 11-12, 2004, Malaga, Spain, 215-226.

[Forsell05a] M. Forsell, Step Caches—a Novel Approach to Concurrent Memory Access on Shared Memory MP-SOCs, In the Proceedings of the 23th IEEE NORCHIP Conference, November 21-22, 2005, Oulu, Finland, 74-77.

[Forsell05b] M. Forsell, Realizing constant time parallel algorithms with active memory modules, International Journal of Electronic Business 3, 3-4 (2005), 255-263.

[Forsell05c] M. Forsell, ParLe—A Parallel Computing Learning Set for MPSoCs/NOCs, In the Proceedings of the International Symposium on System-on-Chip 2005, November 15-17, 2005, Tampere, Finland, 90-95.

[Forsell05d] M. Forsell, Parallel Application Development Scheme for General Purpose NOCs, In the proceedings of the 2005 ECTI International Conference (ECTI-CON 2005), May 12-13, 2005, Pattaya, Thailand, 819-822.

[Hennessy90] J. Hennessy and D. Patterson, Computer Architecture: A Quantitative Approach, third edition, Morgan Kaufmann Publishers Inc., Palo Alto, 1990.

[ITRS05] International Technology Roadmap for Semiconductors, Semiconductor Industry Assoc., 2005; http://public.itrs.net/.

[Jaja92] J. Jaja: Introduction to Parallel Algorithms, Addison-Wesley, Reading, 1992.

[Keller01] J. Keller, C. Keßler, and J. Träff: Practical PRAM Programming, Wiley, New York, 2001.

[Sankaralingam03] K. Sankaralingam, R. Nagarajan, H. Liu, C. Kim, J. Huh, D, Burger, S. Keckler and C. Moore, Exploiting ILP. TLP, and DLP with the polymorphous TRIPS architecture, IEEE Micro 23, 6 (November-December 2003), 46-51.

[Taylor02] M. Taylor, et. al., The Raw Microprocessor: A Computational Fabric for Software Circuits and General Purpose Programs, IEEE Micro 22, 2 (March-April 2002), 25-35.