3

TOTAL ECLIPSE—An Efficient Architectural Realization of the Parallel Random Access Machine

Martti Forsell
Platform Architectures
VTT
Finland

1. Introduction

In the beginning of this millennium power density and related heating problems practically stopped the exponential frequency increase of single core processors and limited availability of *instruction-level parallelism* (ILP) in general purpose applications started to limit the speedup achievable by increasing the number of simultaneously executed instructions in superscalar processors that along with architectural improvements in exploitation of memory hierarchies used to roughly duplicate the performance of processors in every second year for decades. In order to be able to continue the increasing trend of computational performance, all major processor manufacturers have switched to *chip multiprocessors* (CMP) integrating multiple processor cores on a single chip and switching the focus of parallelism from ILP to *thread-level parallelism* (TLP), because the number of transistors per chip still tends to increase exponentially with every new generation of silicon technology (ITRS, 2007) and high amounts of TLP is easier to extract than ILP. Manufacturers have ambitious plans to continue this development by roughly duplicating the number of cores per chip every second year, resulting to constellations with over 100 cores in ten years (Intel, 2006). This will, however, not happen without problems, because current CMP architectures and related programming models do not support simple migration to parallel computing, so called automatic parallelization of existing sequential code has been turned out to be extremely difficult for general purpose programs, writing explicitly parallel versions of programs has turned out to be tedious, error-prone and expensive, and achieving linear speed-ups with respect to the number of cores appears to be limited to only small classes of well-behaving algorithms. These problems are caused by inability of current architectures to hide the latency of shared memory accesses (or intercommunication), lack of synchronicity in execution of computational threads as well as too weak models and low-level primitives of parallel computing forcing a programmer to explicitly take care of data partitioning to maximize locality, functionality mapping supporting data partitioning, synchronization of subtasks, and communication. Without solving these problems, it is hard to imagine that parallel computing would be able to

supersede sequential computing from being the main paradigm of general purpose computing. Furthermore, if nothing is done, the performance of future processors will remain the same while the utilization of processor cores for single computational problems will decrease as the number of cores per chip increases.

The importance of providing easy-to-use programming models has been discovered in parallel computing research long before the era of CMPs (Schwarz, 1966; Karp and Miller, 1969). The culmination of this early active research period was achieved with the invention of the *parallel random access machine* (PRAM) in the late 70's being able to abstract the essence of parallel computing into a conceptually simple and beautiful model being a logical extension the widely used model of sequential computation (Fortune and Wyllie, 1978). A PRAM consists of a set of processors working under the same clock and a uniform single step accessible shared memory connected to them (see Figure 1). Programming with the PRAM model is much easier than with the weaker asynchronous models since with PRAM a programmer knows all the time the exact state of the threads due to synchrony of instruction execution, partitioning and mapping problems are eliminated—a programmer can just put all the data requiring interaction to the shared memory so that all processors can uniformly access it—and communication happens simply via accessing synchronously shared variables in the shared memory. One clear evidence for this is that there exists a rich theory of algorithms for the PRAM model (Jaja, 1992; Keller et al., 2001), which can not be said for the other models that are typically asynchronous and highly architecture dependent. Unfortunately, realization of a computer supporting the PRAM model has turned out to be very challenging. Namely, in our early research (Forsell, 1994) we have shown that the direct implementation of the multiport memory being the key to PRAM implementation is not physically feasible with the known silicon technology if the number of ports is higher than, say 4, due to quadratic wiring area increase with respect to the number of ports. An indirect implementation, based on executing multiple threads per processor core to hide the latency of the memory system, high-bandwidth intercommunication network with randomization to avoid congestion, and wave-based synchronization mechanism, is known from the early 90's (Ranade, 1991), but so far the proposed architectures (Schwarz, 1980; Ranade et al., 1987; Alverson et al., 1990; Abolhassan et al., 1993; Imai, et al., 2000; Vishkin et al., 2008) have been unable to provide feasibility, scalability, instruction-level parallelism (ILP) support, low thread-level parallelism (TLP) support, and cost-efficiency to lure processor manufacturers to employ them in their products.

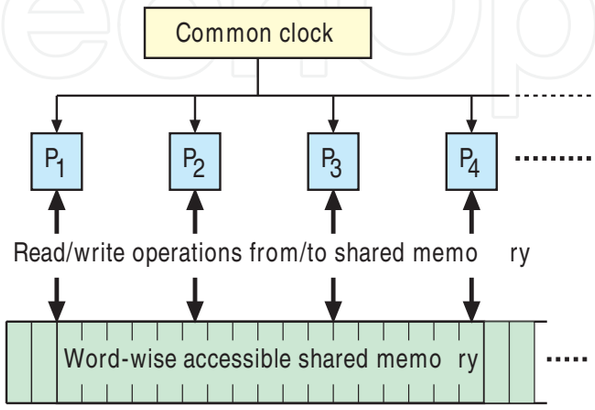

Fig. 1. Parallel random access machine.

In this chapter, we introduce a configurable chip multiprocessor architecture, TOTAL ECLIPSE, for realizing one of the most powerful PRAM variants, the *arbitrary multioperation concurrent read concurrent write* (MCRCW) PRAM model. In addition to standard *arbitrary concurrent read concurrent write* (CRCW) PRAM capable of concurrent reads and writes so that in the case of a write arbitrary of the participating threads succeeds, MCRCW provides multioperations that can e.g. sum the values sent by all participating threads into a memory location concurrently. The architecture is optimized for efficient execution of programs containing enough TLP to hide the latency of the intercommunication network and co-exploitation of virtual ILP with TLP but it is also able to execute programs with low TLP efficiently by providing seamless configurability of PRAM threads to *non-uniform memory access* (NUMA) (Swan et.al., 1977) bunches combining the computational power of two or more threads within a processor core. We will describe the principles of PRAM realization, integration of NUMA bunching to TOTAL ECLIPSE operation, as well as overall architectural structure and operation of the TOTAL ECLIPSE architecture. Performance evaluation by executing simple programs with a clock-accurate simulator is provided and silicon area and power consumption estimations of selected TOTAL ECLIPSE CMP configurations are given. This chapter acts also as a case-driven introduction to novel techniques for parallel architectures, unknown from the theory of sequential architectures.
The rest of the chapter is organized so that in Section 2 we describe the principles of realizing PRAM on a physically feasible silicon platform. In Section 3 we describe the TOTAL ECLIPSE architecture making use of these principles and additional architectural techniques, in Section 4 we evaluate the performance, silicon area and power consumption of selected TOTAL ECLIPSE CMPs, and finally in Section 5 we give conclusions.

2. Realizing the Parallel Random Access Machine

Realizing PRAM on silicon has turned out to be very challenging problem. In addition to the theoretical complexity of direct implementation mentioned in Section 1 (Forsell, 1994), a stronger claim arguing that required bandwidth rules any realization unfeasible was published already in the previous year with the introduction of the LogP model (Culler, 1993). While the complexity of direct implementation can be overcome by using an indirect implementation technique reported a few years earlier (Valiant, 1990; Ranade, 1991), the latter claim has been controversial from the very beginning. The tremendous progress in VLSI technology currently allowing for more than billion transistors and ten on-chip wiring layers with wiring pitch of only 45 nm has raised the capacity and practically achievable bisection bandwidth of a single microchip to a level where these old capacity/bandwidth precautions do not hold any more. In addition, these numbers are predicted to grow for still more than ten years making even more complex integrated systems feasible (ITRS, 2007). Finally, recent estimations on the area and power, and even FPGA and silicon prototypes of PRAM or PRAM-like CMPs (Vishkin, 2007; Forsell and Roivainen, 2008) prove that PRAM realizations are indeed physically feasible. In this section we describe the principles of realizing the PRAM model as formulated by (Ranade, 1991; Leppänen 1996).
The current approach for advanced CMPs is to use a *cache coherent distributed shared memory* (CC-SM) machine consisting of a number of processor cores with local caches connected to memory modules via an asynchronous communication network (see Figure 2). In order to try to hide the latency of the distributed memory system, caches are being kept coherent

during execution by using a high-speed cache coherence mechanism, usually based on distributed directories (Lenoski, 1992). The problems of CC-SMs are that for general purpose parallel algorithms the cache coherence maintenance traffic consumes already the most of the intercommunication network bandwidth, for demanding memory access patterns caches would need to be multiported, thus non-scalable (Forsell, 1994) or severe performance degrading sequentialization will occur, and for fine-grained parallel functionality the asynchrony of the machine makes programming very difficult. It is hard to solve all these problems together without taking a radically different approach like shared memory emulation connecting a set of processor cores without caches to memory modules via a high-bandwidth synchronous intercommunication network (Ranade, 1991; Leppänen, 1996). In it, the latency is hidden with low-overhead multithreading exploiting slackness of parallel computation, i.e. executing other threads while one is referring the memory in a pipelined way. We call the obtained solution *emulated shared memory* (ESM) machine (see Figure 2). A bit similar cacheless solution is used with some synchronous SIMD and vector machines, but they can not execute code including control parallelism efficiently.

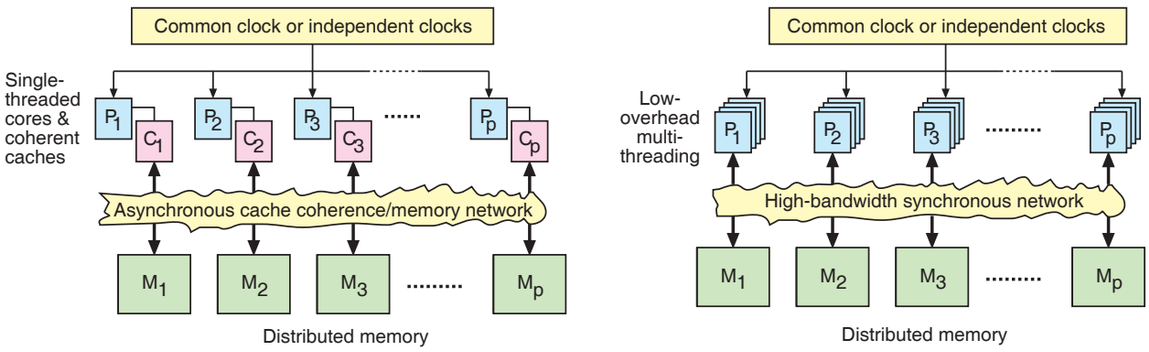

Fig. 2. Cache coherent shared memory (left) versus emulated shared memory approach (right) (P=processor core, C=local cache, M=memory module).

There exists a number of theoretical studies summarized in (Leppänen, 1996) that formally prove that this kind of on ESM can work-optimally simulate the PRAM with a high probability if the following preconditions related to the network topology, and congestion avoidance are guaranteed:

(i) The bandwidth requirements of certain extreme cases causing all the references to be headed to a low number of (or even single) memory module(s) are reduced to an ability to route random traffic by using a hashing of memory locations that is randomly selected from a family of hashings (Dietzfelbinger et.al., 1994).

(ii) To handle random communication the bisection bandwidth of the network must be at least *O(number of cores)*.

(iii) Synchronization of memory references can be handled by the synchronization wave technique that works with acyclic networks in which special synchronization packets are sent by the processors to the memory modules and vice versa (Ranade, 1991). The idea is that when a processor has sent all its packets on their way, it sends a synchronization packet. Synchronization packets from various sources push on the actual packets, and spread to all possible paths, where the actual packets could go. When a node receives a synchronization packet from one of its inputs, it waits, until it has received a

synchronization packet from all of its inputs, then it forwards the synchronization wave to all of its outputs. The synchronization wave may not bypass any actual packets and vice versa. When a synchronization wave sweeps over a network, all nodes and processors receive exactly one synchronization packet via each input link and send exactly one via each output link.

Another necessary condition for practical PRAM implementations is that the used CMP architecture needs to be ultimately implementable with current silicon technology. Due to relatively decreasing signal propagation speed on shrinking silicon technologies, variable link length intercommunication network topologies, including all logarithmic diameter constellations (trees, fat trees, butterflies, hypercubes, etc.) fail to provide performance scalability with respect to the number of processor cores, while fixed link length topologies like coated meshes, sparse meshes and multimeshes have no such scalability problems (Leppänen, 1996; Forsell, 2002; Forsell and Leppänen, 2005).

3. TOTAL ECLIPSE

Embedded Chip-Level Integrated Parallel SupErcomputer (ECLIPSE) is an architectural framework for general purpose chip multiprocessors and multiprocessor systems on chip (MP-SOC), but is extendable also to multichip constellations (Forsell, 2002). It lends many ideas from our early work on the *Instruction-Level Parallel Shared Memory* (IPSM) machine originally reported in (Forsell, 1997) as well as earlier PRAM realization research (Ranade, 1991; Leppänen, 1996) and *network on chip* (NOC) research (Jantsch, 2003). Unfortunately, the original ECLIPSE architecture is only able to support the *exclusive read exclusive write* (EREW) PRAM model which is not able to match the performance of MCRCW PRAM, but requires logarithmically longer execution times for a large number of parallel computational problems even though optimal parallel algorithms are used. In addition, it fails to support efficient execution of low-TLP functionalities because for organizational reasons it features a relatively high minimum number of threads per processor, dropping the utilization of a core to as low as the reciprocal of that value in the case of a functionality having only one thread.

Our renewed proposal for a universal general purpose CMP is the TOTAL ECLIPSE architecture that realizes the arbitrary MCRCW PRAM model and supports NUMA execution for processor-wise thread bunches making execution of low-TLP functionalities as efficient as with standard sequential processors using the NUMA convention. A TOTAL ECLIPSE consists of P T_p-threaded (constituting total $T = PT_p$ threads) F-functional unit MBTAC processor cores with dedicated instruction memory and local data memory modules, P T_p-line step caches and scratchpads attached to processors, P fast data memory modules, and a high-bandwidth multimesh interconnection network (see Figure 3).

In the following subsections we describe the processor, memory system, and communication network of the TOTAL ECLIPSE architecture as well as the key architectural techniques used in them to realize the properties of it. Due to simplicity reasons and lack of space, we limit ourselves to describing an integer-only version of the architecture. Inclusion of floating point support to this class of architectures should be, however, as straightforward as for any other architecture. Supporting application-specific acceleration of functionalities, like graphics, multimedia, and communications, is also left out because they can be implemented efficiently with already relatively well-known architectural solutions that may be used along with TOTAL ECLIPSE, making the overall system architecture slightly

heterogeneous (Forsell, 2009). In such a heterogeneous TOTAL ECLIPSE system, however, the performance of TOTAL ECLIPSE unit in general purpose parallel execution would make useless techniques used in some current heterogeneous systems that map even some general purpose functionality to general purpose GPUs rather than standard multicore CPUs to gain modest speedups (although this happens often with the cost of reduced utilization, increased power consumption, and more difficult programmability).

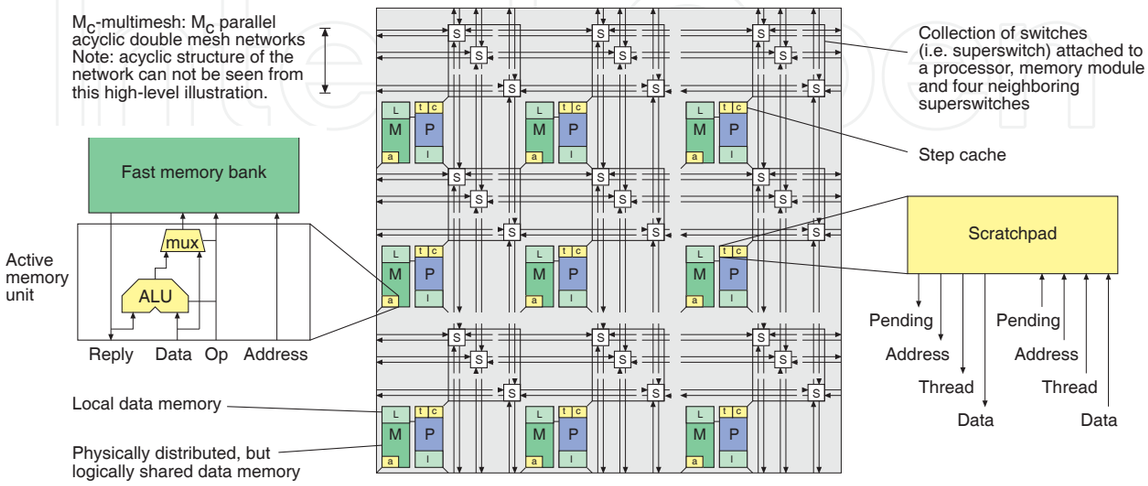

Fig. 3. Block diagram of the TOTAL ECLIPSE architecture (P=processor, M=shared data memory module, L=local data memory module, I=instruction memory module, a= active memory unit, c=step cache, t=scratchpad, and s=switch).

3.1 Processor

Multibunched/threaded Architecture with Chaining (MBTAC) is a dual-mode VLIW processor architecture designed for realizing both a strong PRAM model on a physically distributed memory architecture (so called PRAM mode) and an efficient NUMA model for low TLP locality-optimized code (so called NUMA mode) (Forsell, 2009). An MBTAC processor has *A* ALUs, *M* memory units, *M* hash address calculation units, a compare unit, a sequencer, and a register file of *R* registers per thread on a deep, cyclic, hazard-free interthread pipeline for the PRAM mode execution and a local ALU, a local memory unit, a local sequencer, and a register file of *R* registers per thread bunch on a four stage pipeline for the NUMA mode execution (see Figure 4). The NUMA mode pipeline is overlapped/merged with the first four stages of the PRAM mode pipeline so that most of the hardware, including one ALU and all registers, can be shared between the modes. Other parts of the processor include a step cache and scratchpad that are used to implement concurrent memory access and multioperations. MBTAC has a VLIW-style instruction set with a chain-like fixed execution ordering of subinstructions with a mechanism for using the result of a subinstruction as an operand of the following subinstructions in the chain for the PRAM mode and standard parallel organization of functional units for the NUMA mode (see Appendix A for the list of subinstructions). There is a hardware assisted synchronization mechanism for a limited number of concurrent fast barriers, while a bit slower software based solution utilizing multioperations can be used to provide an arbitrary number of simultaneous barriers (Forsell, 2006).

MBTAC supports overlapped execution of a variable number of threads and thread bunches and seamless dynamic switching between them with special instructions. Multithreading is implemented as a T_p-stage, cyclic interthread pipeline for hiding the latency of the memory system and maximizing the overlapping of execution in the PRAM mode. Switching between threads and bunch slots happens in zero time, because threads proceed in the pipeline only during the forward time. If a thread tries to refer memory when the intercommunication network is busy, the whole pipeline is suspended until the network becomes available again. After issuing a memory read, the thread can wait the reply for at most $M_w<T_p$ clock cycles before the pipeline freezes until the reply arrives. For the NUMA mode, forwading is used to reduce the number of pipeline hazards to two delay slots per each executed control transfer instruction.

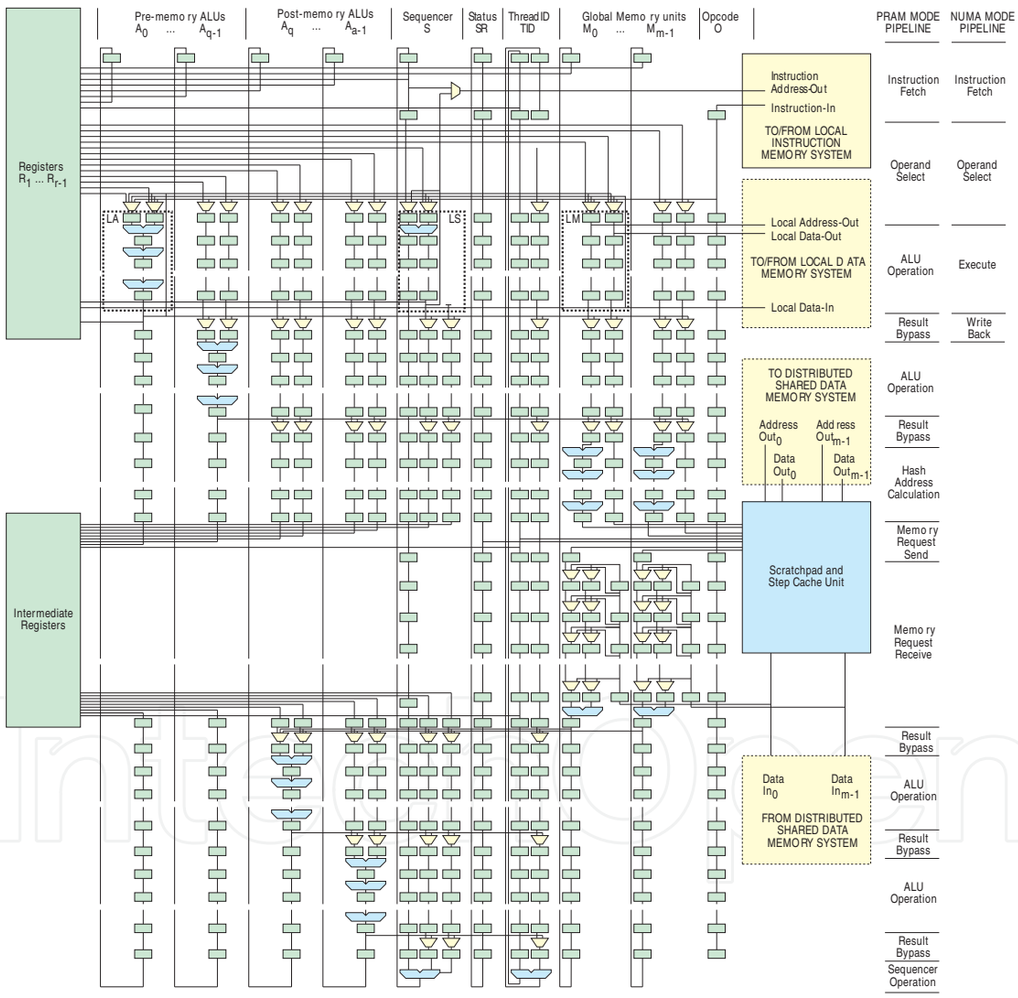

Fig. 4. Block diagram of the MBTAC processor

The PRAM and NUMA models are linked to the architecture so that a full cycle in the pipeline corresponds typically to a single PRAM step and a full cycle of execution for a bunch with B thread slots corresponds typically to executing B consecutive instructions. During a step, each thread of each processor of the CMP executes an instruction, including

at most M shared memory reference subinstructions, and sends a synchronization wave. Therefore a step lasts for multiple, at least T_p+1, clock cycles.
In the following subsections we take a detailed look at special architectural techniques, chaining, step caches, and scratchpads, used in TOTAL ECLIPSE.

3.1.1 Low and low-level parallelism exploitation via chaining and bunching

The organization of the PRAM mode functional units in MBTAC is targeted for exploiting ILP during steps of parallel execution. Therefore functional units in MBTAC are connected as a chain, so that a unit is able to use the results of its predecessors in the chain (Forsell, 1997; Forsell, 2003). Since multiple threads are executed in an overlapped way, it possible to execute dependent subinstructions during a step unlike with parallel functional unit organization of sequential processors (see Figure 5). We call this new class of parallelism *virtual instruction level parallelism*. In order to maximize the obtained speedup, the ordering of functional units in the chain is selected according to the average ordering of instructions in a basic block: Two thirds of the ALUs form the beginning of the chain. They are followed by the memory units and the rest of the ALUs. The compare unit and the sequencer are located in the end of the chain, because comparing and branching happen always in the end of basic blocks. In the NUMA mode, the local functional units are organized in parallel like in a standard single threaded VLIW processor because chaining would cause a lot of pipeline hazards for bunches and actually degrade the performance.

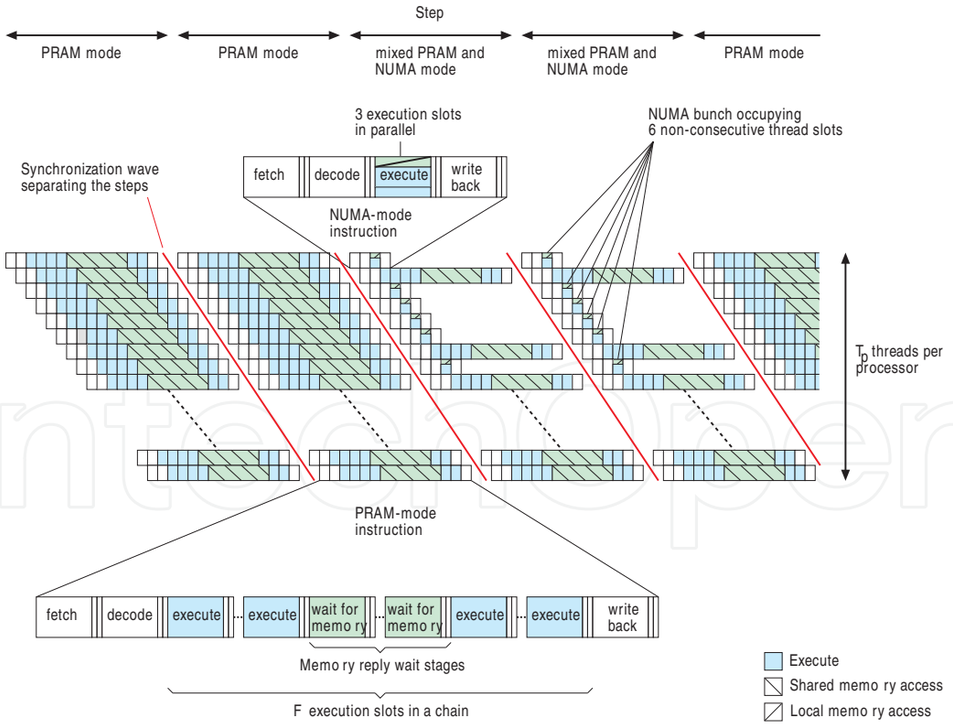

Fig. 5. Chaining and bunching.

Efficient execution of low TLP code is implemented by making the thread storage configurable/indirect and pipeline suitable for sequential execution so that multiple thread

execution slots can be assigned to efficiently execute a single NUMA mode thread bunch by just using the same thread storage address for all of them (Forsell, 2009). This way a bunch can use thread slots to execute multiple instructions during a step removing the low TLP performance bottleneck of the original Eclipse (see Figure 5). The number of concurrent bunches per processor can be everything from zero (PRAM mode) to $T_p/2$ and they can occur in parallel with PRAM mode threads. Bunches can only access local memories since there is no efficient and easy-to-use mechanism to hide the latency of memory references in low TLP situations. Required indirect thread storaging is implemented by storing threads into a multiported and multithreaded register block (like in the SUN Sparc Tx-series) rather than in the pipeline registers, and by adding a thread address storage pointer for each thread (see leftmost registers of the TID dual chain in Figure 4). In order to set a group of threads to use just one thread storage, i.e. to execute a single thread for all the thread slots, a programmer needs just to set the thread storage pointers to a single value selected out of the values of the thread storage pointers with the JOIN instruction. Similarly, splitting the bunch back to separate threads happens by restoring the old numbering of the thread slots with the SPLIT instruction.

3.1.2 Concurrent access and step caches

The PRAM support machinery of TOTAL ECLIPSE allows for 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. This is implemented by using step caches, which are associative memory buffers in which data stays valid only to the end of ongoing step of multithreaded execution (Forsell, 2005). The main contribution of step caches to concurrent accesses is that they step-wisely filter out everything but the first reference for each referenced memory location. This reduces the number of requests per location to P allowing them to be processed sequentially on a single ported memory module assuming $T_p \geq P$ (see Figure 6).

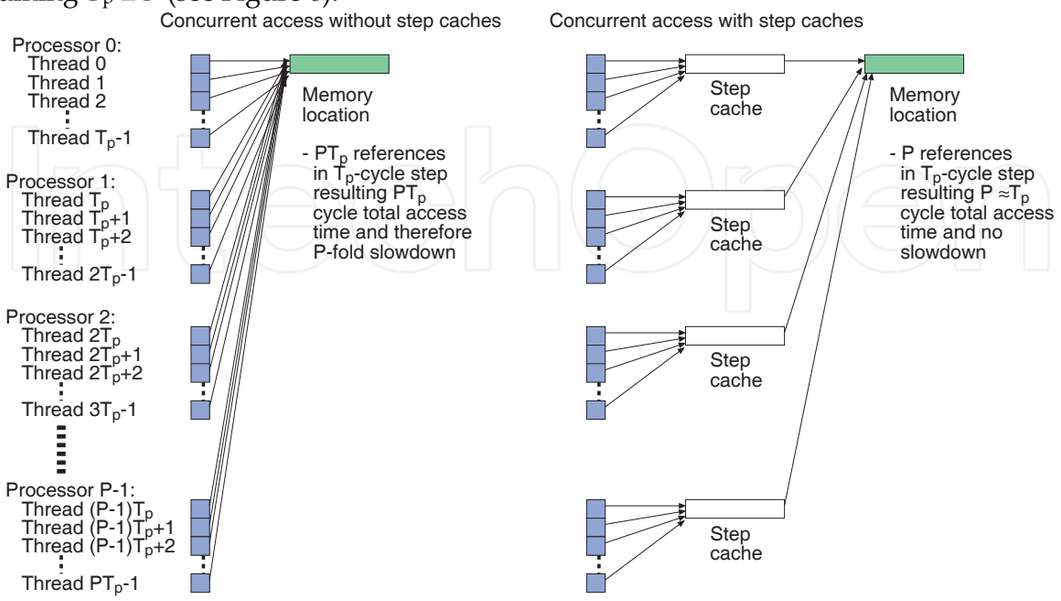

Fig. 6. Step caches for implementing concurrent memory access.

Step caches operate similarly as ordinary caches with a few notable exceptions: Each time a multithreaded processor refers to the shared data memory a step cache search is performed. A hit is detected on a cache line if the line is in use, the address tag matches the tag of the line, and the least significant bits of step of the reference matches the step of the line. In the case of a hit, a write is just ignored while a read is just completed by accessing the data from the cache. In the case of a miss, the reference is stored into the cache using the replacement policy at hands and marked as pending (for reads). At the same time with storing the reference information to the cache line, the reference itself is sent to the lower-level memory system. When a reply of a read arrives from the memory, the data is put to the data field of the line storing the reference information and the pending field is cleared. The structure of a step cache is similar to ordinary caches, but it has two extra fields—pending and step—and a block for decaying (Kaxiras, 2001) the data belonging to previous steps before their step field matches again to the least significant bits of current step (see Figure 7). Cache coherency problems are avoided due to a short life-time of references in the cache, since operations made during a step are independent by the definition parallel execution. The TOTAL ECLIPSE CMPs involved in our evaluations in Section 4 use A_s -way set associative step caches with *the least recently used* (LRU) replacement policy of size T_p lines attached to each processor and scratchpads.

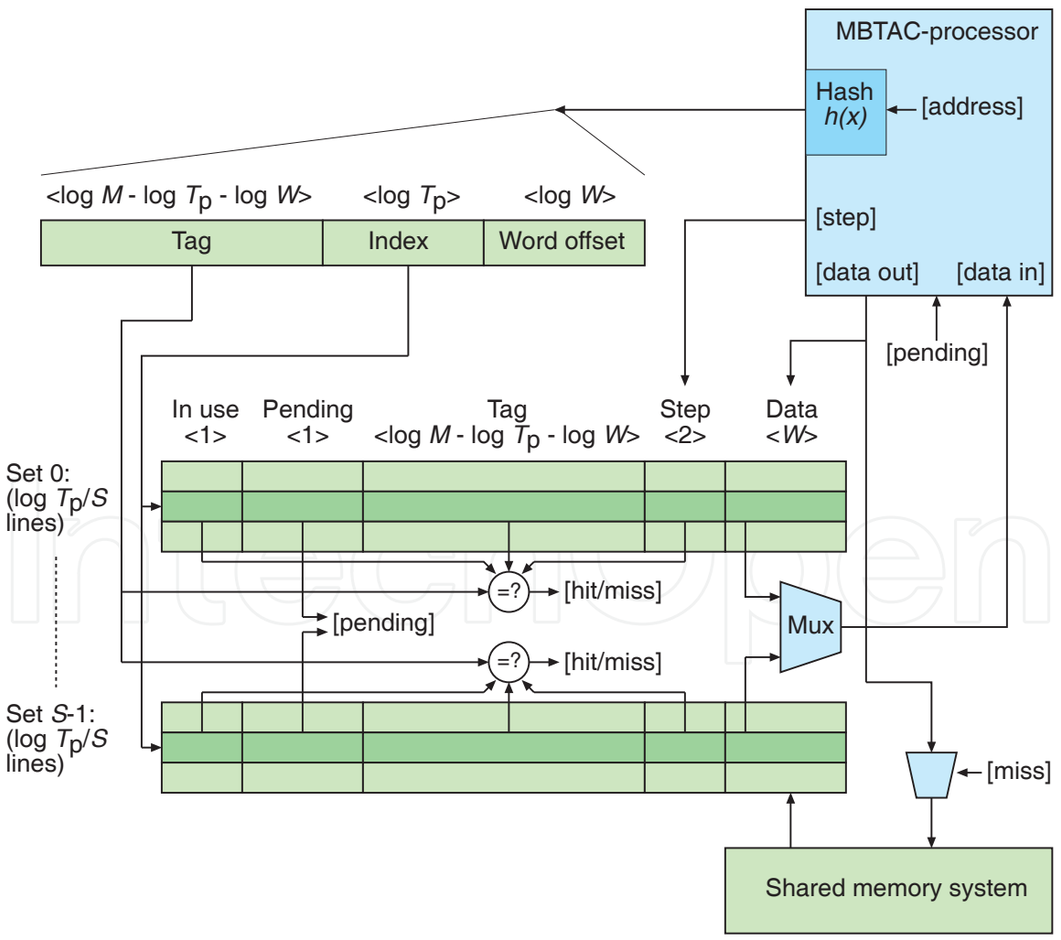

Fig. 7. Organization of an A_s -way associative step cache.

3.1.3 Multioperations and scratchpads

Scratchpads are addressable memory buffers that are used to store memory access data to keep the associativity of step caches limited in implementing multioperations and thread bunches with a help of step caches, and minimal on-core and off-core ALUs that take care of actual intra-processor and inter-processor computation for multioperations (Forsell, 2006) (see Figures 3 and 4). Scratchpads are organized with step caches to so called scratchpad - step cache units. A scratchpad - step cache unit for MBTAC processor consists of a T_p-line scratchpad, a T_p-line step cache, and a simple multioperation ALU for executing incoming concurrent references, multioperations and arbitrary ordered multiprefixes sequentially (see Figure 8).

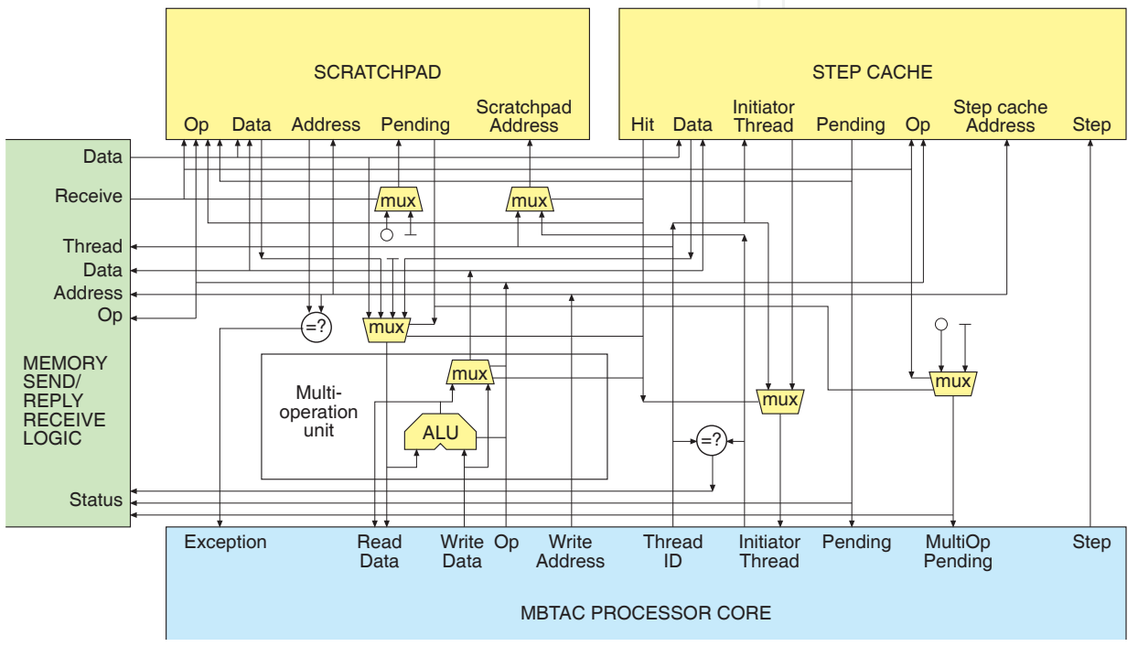

Fig. 8. Implementation of multioperations with scratchpads and step caches. Detailed description of this logic can be found in (Forsell, 2006).

Ordinary multioperations are implemented as two consecutive single step operations (see Appendix A for a list of available multioperations). During the first step, a starting operation (BMxx for multioperations or BMPxx for arbitrary ordered multiprefix operations) executes a processor-wise multioperation against a step cache location without making any reference to the external memory system (see Figure 9). During the second step, an ending operation (EMxx for multioperations or EMPxx for arbitrary ordered multiprefix operations) performs the rest of the multioperation so that the first reference to a previously initialized memory location triggers an external memory reference using the processor-wise multioperation result as an operand. The external memory references that are targeted to the same location are processed in the active memory unit of the corresponding memory module according to the type of the multioperation. In the case of arbitrary ordered multiprefixes the reply data is sent back to scratchpads of participating processors. The consecutive references are completed against the step cached reply data. It can happen that a consecutive reference is made to a location while the external reference is being processed.

In that case, the operation is marked as pending and completed as the result is available. This does not slow down the processing any way since one additional simple ALU is located to the end of memory access pipeline segment in MBTAC (see Figure 4). Since MBTAC uses limited associativity step caches, scratchpads are used to store the id of the initiator thread of each multioperation sequence to the step cache and internal initiator thread id (IT) register as well as reference information to a storage that saves the information regardless of possible conflicts that may wipe away information on references from the step cache. A scratchpad has a field for data, address and pending for each thread of the processor. With a help of scratchpads, multioperations are implemented by using sequences of two instructions: Data to be written in the step cache is also written to the scratchpad, id of the first thread referencing a certain location is stored to the step cache and IT register (for the rest of references), the pending bit for multioperations is kept in the scratchpad rather than in the step cache, reply data is stored to the scratchpad rather than to the step cache, and reply data for the ending operation is retrieved from the scratchpad rather than from the step cache (Forsell, 2006).

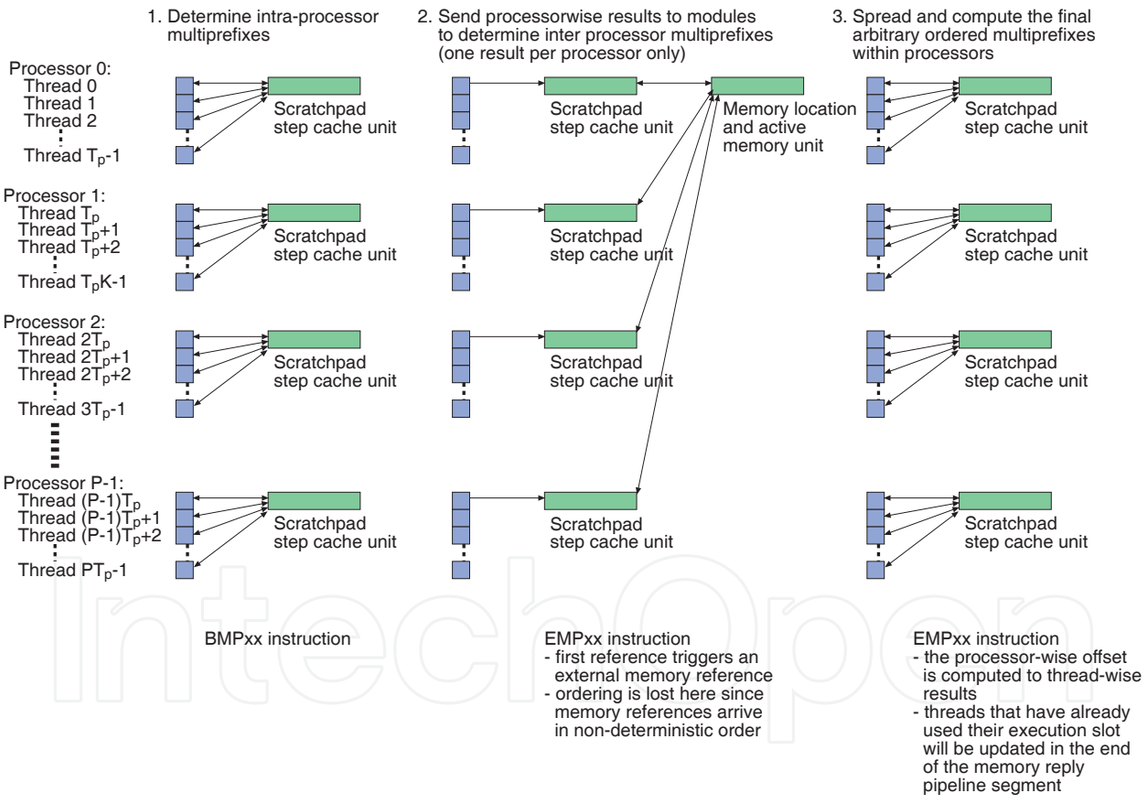

Fig. 9. Implementation of multioperations with scratchpads and step caches.

Since many efficient parallel algorithms make use of limited concurrent access, constituting of, say, at most square root T references per step, we have implemented faster single instruction limited multioperations that execute in single step. These instructions do not use multioperation units of processors but just active memory ALUs to perform their operations.

3.2 Memory modules

Total ECLIPSE has three types of memory modules—local data memory modules, shared data memory modules, and instruction memory modules. For performance reasons, they are accessed via dedicated local data, shared data, and instruction memory ports of processors, respectively (see Figure 10). The local memory modules are aimed for storing data local to threads of a processor and NUMA mode data while all the shared data is located to distributed shared data memory modules emulating the ideal PRAM memory. Instruction memory modules are aimed to keep the program code for each processor. The modules are connected together so that all memory locations can be accessed via the shared data memory port but giving high priority to accesses from local data memory and instruction memory ports (see Figure 10).

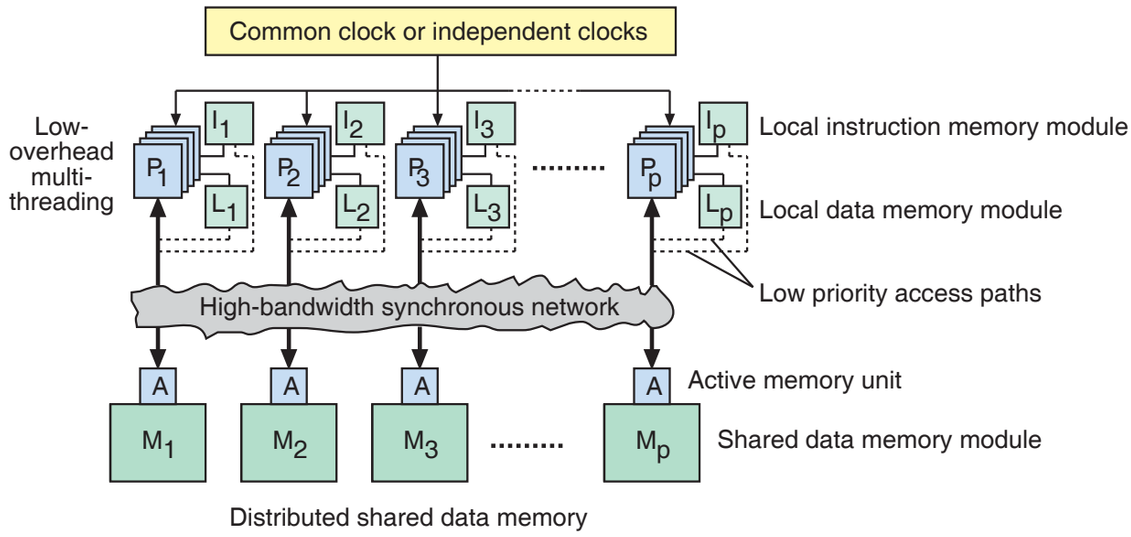

Fig. 10. Organization of the memory system

During normal operation, the on-chip shared data, local data, and instruction memory modules are isolated from each other to guarantee high-bandwidth local data, shared data, and instruction streams to processors. The access (and cycle) times of local data and instruction modules equal to one system clock cycle. The access time of shared data modules need to be half of the system clock cycle or alternatively T_p must be at least $2P$ or a small and fast module-level cache (allowing for multioperation related data to be read and written during a single clock cycle) is needed for each memory module. A local data memory module is just a standard memory module. A shared data memory module consists of an active memory unit and data memory itself (see Figure 3). An active memory unit consists of a simple ALU and fetcher (Forsell, 2006). Active memory units allow one to perform arbitrary ordered multiprefix operations and multioperations that e.g. sum all the references that are targeted to a memory location during a step helping to drop the lower bound of the execution time of some parallel algorithms by a logarithmic factor and perform flexible synchronizations (including arbitrary number of simultaneous barriers) between threads. Instruction memory modules are similar to data memory modules except they do not have active memory units, the length of instruction words is different to that of data words depending on the architectural parameters, and there are no write lines from the

instructions fetcher to instruction memory modules. If the data or program code of the application does not fit into the on-chip memory, expensive external memory access prefetches with interleaving, banking and module-level caching are needed. In this chapter, however, we consider on-chip memory configurations only.

3.3 Interconnection network

The TOTAL ECLIPSE network is a M_c-way double acyclic two-dimensional multi mesh (Forsell and Leppänen, 2005) (see Figure 11). It has separate lines for references going from processors to memories and for replies from memories to processors to maximize the throughput for read-intensive portions of code. Memory locations are distributed across the data modules by a randomly chosen polynomial hashing function for avoiding congestion of messages and hot spots (Ranade, 1991; Dietzfelbinger et.al., 1994). References are routed by using a simple greedy algorithm on a randomly selected submesh. Deadlocks are not possible during communication because the network is acyclic. Separation of steps and their synchronization is guaranteed with the synchronization wave technique allowing for independent clocking or asynchronous links between the processor cores.

To exploit locality, the switches related to processor-memory module pairs are grouped as superswitches (see Figure 11). This kind of a two-level structure allows for sending a message from a resource to any of the switches belonging to a superswitch in a single clock cycle. A superswitch consists of M_c switches that are connected to a processor and memory module via dedicated output decoders and switch elements. Each switch consists of 8 switch elements that have two to three input and output links. A switch element consists of logic blocks for determining the right output link (select direction), arbitration logic, and output queues storing the outgoing messages (see Figure 11). A switch element routes an incoming message to an output buffer according to the target information of the message if there is room for it in the buffer. If multiple incoming messages need to be routed to a single output buffer simultaneously it is waited until there is room in the buffer for all of them before transferring them simultaneously to the output buffer. If an incoming message is not allowed to proceed to the output buffer, the busy signal is activated in the corresponding input.

The processors send memory requests (reads and writes) and synchronization messages to the memory modules and modules send replies and synchronization messages back to processors. A message is built of a single parallel flit consisting of dedicated fields for message type, data access width, target address, return address and data (Forsell, 2005). Messages are routed at the rate of at most one hop per clock cycle by using a simple greedy algorithm with two intermediate targets (see Figure 11): A message is first sent to a first intermediate target, which is a randomly chosen switch in a superswitch related to the sending resource (this determines the submesh to be used for routing). Then the message is routed greedily (go to the right row and then go to the right column) to the second intermediate target, which is the switch of the selected submesh in the superswitch related to the target resource. Finally the message is routed from the second intermediate target to the target resource. Routing memory replies back to the processors is made in the same way, but using the memory reply network. Synchronization messages follow the same paths from processors to memories and back to processors.

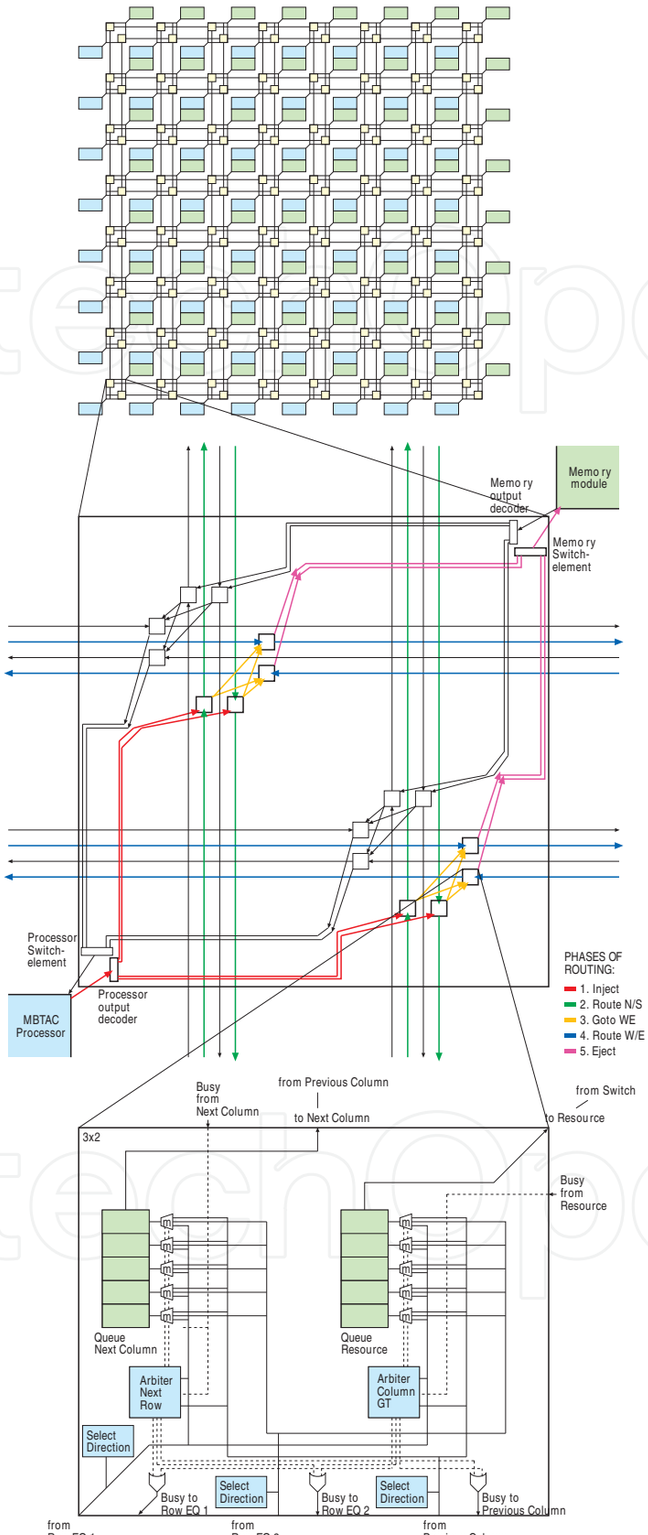

Fig. 11. Block diagrams of a M_c-way double acyclic multimesh network (top), superswitch (middle), and switch element (bottom) for a 64-processor TOTAL ECLIPSE CMP.

4. Evaluation

In order to evaluate the performance and scalability achievable with the TOTAL ECLIPSE architecture on realistic and physically feasible CMPs we made a number of simulations on different CMP configurations and estimated the silicon area and power consumption of the used configurations with analytical modeling.

For performance tests, we mapped parallel and sequential e-language versions of seven parallel computational problems of which three are fixed size and others depend on the number of threads in a processor core (see Table 1) to PRAM thread groups and NUMA bunches, compiled, optimized (e-compiler options -O2 -ilp -fast) and loaded them to three CMP configurations having 4, 16 and 64 ten-FU 512-threaded MBTAC processors (see Table 2), and executed them with our clock accurate CMP simulator modified for the TOTAL ECLIPSE architecture.

In order to evaluate the PRAM mode execution performance, we executed the parallel versions of the programs in the TOTAL ECLIPSE CMPs in the PRAM mode and in ideal PRAMs having similar configurations. The results as relative execution time are shown in Figure 12. We can observe that the PRAM mode execution speed of TOTAL ECLIPSE is very close to that of ideal PRAM, mean overheads being 0.8%, 1.7%, and 1.4% for E4, E16, and E64, respectively.

	SEQUENTIAL				**PARALLEL**		
Name	***N***	***E***	***P***	***W***	***E***	***P=W***	**Explanation**
aprefix	*T*	*N*	1	*N*	1	*N*	Determine an arbitrary ordered multiprefix of an array of *N* integers
fft	64	$N \log N$	1	$N \log N$	1	N^2	Perform a 64-point complex Fourier transform using fixed point arithmetic on integer ALUs
max	*T*	*N*	1	*N*	1	*N*	Find the maximum of a table of *N* words
mmul	16	N^3	1	N^3	1	N^3	Compute the product of two 16-element matrixes
sort	64	$N \log N$	1	$N \log N$	1	N^2	Sort a table of 64 integers
spread	*T*	*N*	1	*N*	1	*N*	Spread an integer to all *N* threads
sum	*T*	*N*	1	*N*	1	*N*	Compute the sum of an array of *N* integers

Table 1. Evaluated computational problems and features of their sequential and parallel implementations (*E*=execution time, *M*=size of the key string, *N*=size of the problem, *P*=number of processors, *T*=number of threads, *W*=work). Note that fft, mmul, and sort are fixed size problems, while others depend on *T*.

	Symbol	E4	E16	E64	DLX
Model of computing	M_{tlp}	PRAM / NUMA	PRAM / NUMA	PRAM / NUMA	RAM
ILP model in the PRAM mode	M_{ilpp}	chained VLIW	chained VLIW	chained VLIW	
ILP model in the NUMA mode	M_{ilpn}	VLIW	VLIW	VLIW	5-stage pipelin e
Processors	P	4	16	64	1
Threads per processors	T_p	512	512	512	1
Total number of threads	T	2048	8192	32768	1
FUs in the PRAM mode	F_p	10	10	10	-
FUs in the NUMA mode	F_n	3	3	3	4
On-chip shared data memory	M_{sd}	2 MB	8 MB	32 MB	-
On-chip local data memory	M_{ld}	2 MB	8 MB	32 MB	-
On-chip banks access time	A_b	1 c	1 c	1 c	1 c
On-chip bank cycle time	C_b	1 c	1 c	1 c	1 c
Length of FIFOs	Q	16	16	16	
Step cache associativity	A_c	4	4	4	-

Table 2. Evaluated configurations (c=processor clock cycles). DLX is a single threaded RISC processor described in (Hennessy and Patterson, 2003). The Random Access Machine (RAM) model is a computing model used in sequential computers.

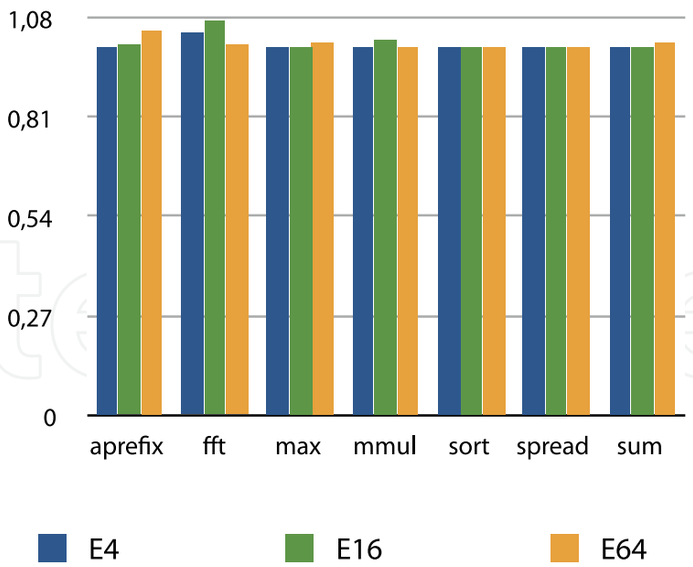

Fig. 12. The relative execution time of TOTAL ECLIPSE CMPs compared to ideal PRAMs with similar configuration (PRAM=1.0, shorter is better).

The NUMA mode performance was measured by executing the sequential versions of the programs in a single thread of a CMP in both PRAM and NUMA modes. In NUMA mode

execution all the threads of a single processor were joined to a single NUMA bunch. The results of these simulations as execution time are illustrated in Figure 13. We see that the NUMA mode indeed provides better performance for sequential programs than the PRAM mode, but is not able to exploit virtual ILP up to degree possible in the PRAM mode. The mean speedups of using the NUMA mode are 13200%, 13196%, and 13995% for E4, E16, and E64, respectively. This does not, however, mean that these NUMA bunches can solve these computational problems faster than the PRAM mode if parallel solutions are used. Namely, the parallel solutions are 1421%, 3111%, and 6889% faster than the best sequential ones for E4, E16, and E64, respectively. Note that the speedup is not linear with respect to the number of processors, since 3 out of 7 benchmarks are fixed size computational problems.

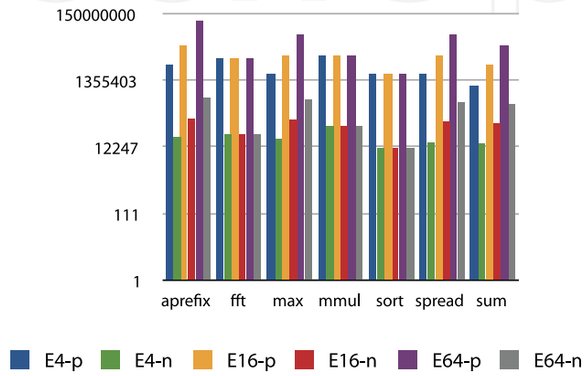

Fig. 13. The execution time of sequential solutions of the computational problems on a single thread of a singe MBTAC processor core in the PRAM mode and on a 512-thread NUMA bunch in a single MBTAC processor core.

To show seamless configurability between NUMA and PRAM modes in the TOTAL ECLIPSE architecture, we measured the NUMA mode execution time for sort algorithm for a bunch with different number of threads ranging from 1 to 512 threads per bunch in the E4 configuration. The results are shown in Figure 14. We can see linear performance increase as the number of threads per the bunch increases (note that the thread scale is exponential).

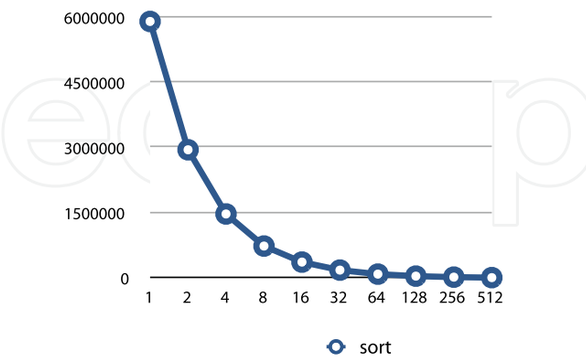

Fig. 14. Execution time of as a function of number of threads in the bunch for E4 CESM configuration.

We compared also the NUMA mode performance of TOTAL ECLIPSE CMPs to that of a single threaded five-stage basic pipelined RISC processor DLX (Hennessy and Patterson,

2003) by executing all the sequential programs in a single DLX processor with a single step accessible on-chip memory (like the local memories of TOTAL ECLIPSE cores) and in a single NUMA bunch composed of the threads of a single processor of TOTAL ECLIPSE. In order to commit fair comparison, we took the variable size of the problems aprefix, max, spread, and sum into account in our measurements so that the amount of actual computation (and the computational problem itself) is the same for the both architectures. In addition, the same compiler and even compilation were used to eliminate the effect of the compiler. TOTAL ECLIPSE code was obtained from DLX code just by doing binary translation (Forsell, 2003). The results are shown in Figure 15. Although the code is not optimized with a VLIW compiler for TOTAL ECLIPSE's NUMA bunching, it provides a bit better performance than DLX, the average speedup being 8.8%. This is due to more efficient ILP architecture of TOTAL ECLIPSE cores.
Finally, we estimated silicon area, power consumption, and maximum clock frequency figures for E4, E16, and E64 with configurable memory modules implemented on a high-performance 65 nm silicon process. The estimations are based on models presented (Pamunuwa et. al., 2003), ITRS 2007, and careful counting of architectural elements broken down to gate counts. The wire delay model gives maximum clock frequency 1.29 GHz for E4, E16 and E64 assuming 135 nm global interconnect wiring with repeaters. The area and power results are shown in Figure 16. These figures except the clock frequency are somewhat comparable to those of a X86 class multi-core high-frequency superscalar processor.

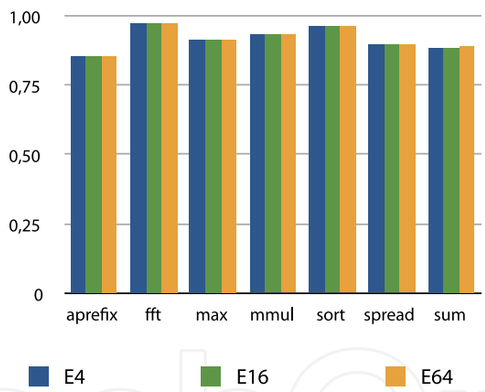

Fig. 15. Relative execution time of 512-thread NUMA bunches compared to 5-stage pipelined DLX processor with the same memory setup (DLX=1.0, shorter is better).

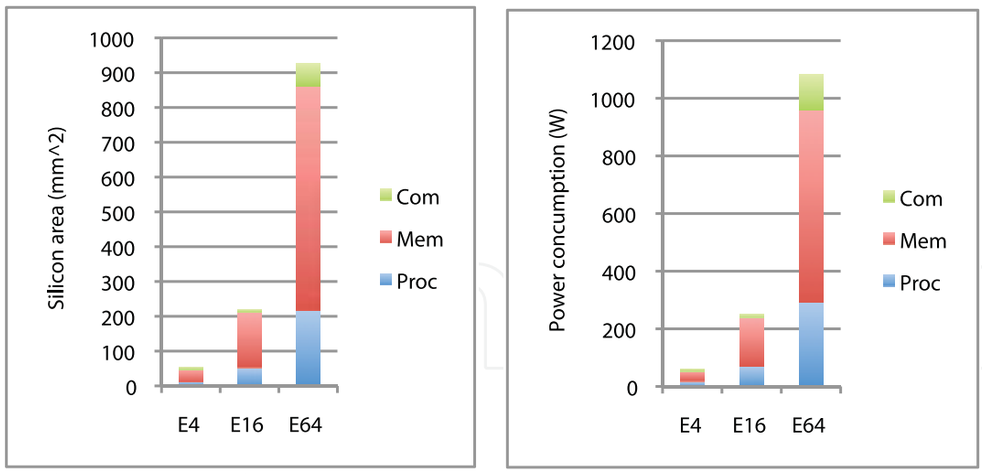

Fig. 16. Silicon area and power consumption estimates for E4, E16, and E64 with configurable memory module at 1.29 MHz on a high-performance 65 nm technology (Com=communication network, Mem=memory modules, and Proc=processors).

5. Conclusion

We have introduced the TOTAL ECLIPSE CMP architecture providing an efficient realization of PRAM. In addition to providing synchronous access to the shared memory, it allows for concurrent references to memory location, special multioperations performing computations between the participating threads, modes for efficient parallel execution and fast sequential operation combining the computational power of threads and seamless configurability between these modes. According to our evaluation TOTAL ECLIPSE provides in many cases performance close to similarly configured ideal PRAM, while the silicon area and power consumption are somewhat comparable to the current commercial CMPs. This chapter acts also as a case-driven introduction to novel parallel architecture techniques, including synchronization wave, cacheless memory organization, chaining, step caching, bunching, and scratchpads, that are unknown from the theory of sequential architectures. Our future research interests related to this topic include building FPGA and silicon prototypes of TOTAL ECLIPSE, addressing the off-chip memory efficiency problem, as well as investigating the limits of practical scalability of this kind of architectures.

6. Acknowledgements

This work was supported by the grants 122462 and 128733 of the Academy of Finland.

7. References

Abolhassan, F., Drefenstedt, R., Keller, J., Paul, W. Scheerer, D. (1993) On the Physical Design of PRAMs, *Computer Journal* 36, 8 (1993), 756-762.

Alverson, R., Callahan, D., Cummings, D., Kolblenz, B., Porterfield, A., Smith, B. (1990). The Tera Computer System, *Proceedings of the International Conference on Supercomputing*, Association for Computing Machinery, New York, 1990, 1-6.

Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., von Eicken, T. (1993). LogP: Towards a Realistic Model of Parallel Computation, *Proceedings of the 4th ACM Conference on Principles & Practicies of Parallel Programming*, 1-12.

Dietzfelbinger, M., Karlin, A., Mehlhorn, K., Mayer auf der Heide, F., Rohnert, H., Tarjan, R. (1994). Dynamic Perfect Hashing: Upper and Lower Bounds, *SIAM Journal on Computing* 23, (August 1994), 738-761.

Forsell, M. (1994). Are Multiport Memories Physically Feasible?, *Computer Architecture News* 22, 4 (September 1994), 47-54.

Forsell, M. (1997). Implementation of Instruction-Level and Thread-Level Parallelism in Computers, *Dissertations* 2, Department of Computer Science, University of Joensuu, Joensuu, 1997.

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

Forsell, M. (2003). Using Parallel Slackness for Extracting ILP from Sequential Threads, *Proceedings of the SSGRR-2003s, International Conference on Advances in Infrastructure for Electronic Business, Education, Science, Medicine, and Mobile Technologies on the Internet*, July 28 - August 3, 2003, L'Aquila, Italy.

Forsell, M., Leppänen, V. (2005). High-Bandwidth on-chip Communication Architecture for General Purpose Computing, *Proceedings of the 9th World Multiconference on Systemics, Cybernetics and Informatics (WMSCI 2005)* Volume IV, July 10-13, 2005, Orlando, USA, 1-6.

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

Forsell, M. (2006). Realizing Multioperations for Step Cached MP-SOCs, *Proceedings of the International Symposium on System-on-Chip 2006 (SOC'06)*, November 14-16, 2006, Tampere, Finland, 77-82.

Forsell, M., Roivainen, J. (2008). Performance, Area and Power Trade-Offs in Mesh-Based Emulated Shared Memory CMP Architectures, *Proceedings of the 2008 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'08)*, July 14-17, 2008, Las Vegas, USA, 471-477.

Forsell, M. (2009). Configurable Emulated Shared Memory Architecture for general purpose MP-SOCs and NOC regions, *Proceedings of the 3rd ACM/IEEE International Symposium on Networks-on-Chip*, May 10-13, 2009, San Diego, USA, 163-172.

Fortune, S., Wyllie, J. (1978). Parallelism in Random Access Machines, *Proceedings of 10th ACM STOC*, Association for Computing Machinery, New York, 1978, 114-118.

Hennessy, J., Patterson, D. (2003). *Computer Architecture: A Quantitative Approach*, third edition, Morgan Kaufmann Publishers Inc., Palo Alto, 2003.

Imai, M., Hayakawa, Y., Kawanaka, H., Chen, W., Wada, K., Castanho, C., Okajima, Y., Okamoto, H. (2000). A Hardware Implementation of PRAM and Its Performance Evaluation, *Proceedings of the 15 IPDPS 2000 Workshops on Parallel and Distributed Processing*, May 1-5, 2000, Cancun, Mexico, LNCS 1800, 143 - 148.

Intel. (2006). Research at Intel From a Few Cores to Many: A Tera-scale Computing Research Overview, *White Paper*, Intel, 2006.

ITRS (2007). International Technology Roadmap for Semiconductors, Semiconductor Industry Assoc., 2007; http://public.itrs.net/.

Jaja, J. (1992). *Introduction to Parallel Algorithms*, Addison-Wesley, Reading, 1992.

Jantch, A. (2003). *Networks on Chip* (edited by A. Jantsch and H. Tenhunen), Kluver Academic Publishers, Boston, 2003, 173-192.

Kaxiras, S., Hu, Z. (2001). Cache Decay: Exploiting Generational Behavior to Reduce Cache Leakage Power, *Proceedings of the International Symposium on Computer Architecture*, June 30-July 4, 2001, Göteborg, Sweden, 240-251.

Karp, R., Miller, R. (1969). Parallel Program Schemata, *Journal of Computer and System Sciences* 3, 2 (1969), 147-195.

Keller, J., Keßler, C., Träff, J. (2001). *Practical PRAM Programming*, Wiley, New York, 2001.

Lenoski, D., Laudon, J., Gharachorloo, K., Weber, W., Gupta, A., Hennessy, J., Horowitz, M., Lam, M. (1992). The Stanford Dash Multiprocessor, *IEEE Computer* 25, (March 1992), 63-79.

Leppänen, V. (1996). Studies on the realization of PRAM, *Dissertation* 3, Turku Centre for Computer Science, University of Turku, Turku, 1996.

Pamunuwa, D., Zheng, L-R., Tenhunen, H. (2003). Maximizing Throughput Over Parallel Wire Structures in the Deep Submicrometer Regime, *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 11, 2 (April 2003), 224-243.

Ranade, A., Bhatt, S., Johnson, S. (1987). The Fluent Abstract Machine, *Technical Report Series* BA87-3, Thinking Machines Corporation, Bedford, 1987.

Ranade, A. (1991). How to Emulate Shared Memory, *Journal of Computer and System Sciences* 42, (1991), 307--326.

Schwarz, J. (1966). Large Parallel Computers, Journal of the ACM 13, 1 (1966), 25-32.

Schwarz J. (1980). Ultracomputers, *ACM Transactions on Programming Languages and Systems* 2, 4 (1980), 484-521.

Swan, R., Fuller, S., Siewiorek, D. (1977). Cm*—A Modular Multiprocessor, *Proceedings of NCC*, 645-655, 1977.

Valiant L. (1990). A Bridging Model for Parallel Computation, *Communications of the ACM* 33, 8 (1990), 103-111.

Vishkin, U. (2007). Towards Realizing a PRAM-On-Chip Vision, *Workshop on Highly Parallel Processing on a Chip (HPPC)*, August 28, 2007, Rennes, France (see http://www.hppc-workshop.org/HPPC07/talks.html).

Vishkin, U., Caragea, G., Lee, B. (2008). Models for Advancing PRAM and Other Algorithms into Parallel Programs for a PRAM-On-Chip Platform, *Handbook of Parallel Computing—Models, Algorithms and Applications* (editors S. Rajasekaran and J. Reif), Chapman & Hall/CRC, Boca Raton, 2008, 5-1—5-60.

Appendix A. Core instruction set of TOTAL ECLIPSE

The core instruction set of the integer-only version of the proposed MBTAC processor of TOTAL ECLIPSE consists of an instruction that can be further divided to the A ALU subinstructions, M memory subinstructions, a single compare unit subinstruction, a single sequencer subinstruction, O immediate operand subinstructions, and W_b write back subinstructions in the PRAM mode and to an ALU subinstruction, a memory subinstruction, a sequencer subinstruction, and two write back subinstructions in the NUMA mode. The following list shows the available subinstructions for each class of units:

Memory Unit subinstructions

LDBn	Xx	Load byte from memory n address Xx in MU n
LDBUn	Xx	Load byte from memory n address Xx unsigned in MU n
LDHn	Xx	Load halfword from memory n address Xx in MU n
LDHUn	Xx	Load halfword from memory n address Xx unsigned in MU n
LDn	Xx	Load word from memory n address Xx in MU n
STBn	Xx,Xy	Store byte Xx to memory n address Xy in MU n
STHn	Xx,Xy	Store halfword Xx to memory n address Xy in MU n
STn	Xx,Xy	Store word Xx to memory n address Xy in MU n
MADDn	Xx,Xy	Add multiple Xx to active memory Xy in MU n
MSUBn	Xx,Xy	Subtract multiple Xx to active memory Xy in MU n
MANDn	Xx,Xy	And multiple Xx to active memory Xy in MU n
MORn	Xx,Xy	Or multiple Xx to active memory Xy in MU n
MMAXn	Xx,Xy	Max multiple Xx to active memory Xy in MU n
MMAXUn	Xx,Xy	Max unsigned multiple Xx to active memory Xy in MU n
MMINn	Xx,Xy	Min multiple Xx to active memory Xy in MU n
MMINUn	Xx,Xy	Min unsigned multiple Xx to active memory Xy in MU n
MPADDn	Xx,Xy	Arbitrary multiprefix add Xx to active memory Xy in MU n
MPSUBn	Xx,Xy	Arbitrary multiprefix subtract Xx to active memory Xy in MU n
MPANDn	Xx,Xy	Arbitrary multiprefix and Xx to active memory Xy in MU n
MPORn	Xx,Xy	Arbitrary multiprefix or Xx to active memory Xy in MU n
MPMAXn	Xx,Xy	Arbitrary multiprefix max Xx to active memory Xy in MU n
MPMAXUn	Xx,Xy	Arbitrary multiprefix max unsigned Xx to active memory Xy in MU n
MPMINn	Xx,Xy	Arbitrary multiprefix min Xx to active memory Xy in MU n
MPMINUn	Xx,Xy	Arbitrary multiprefix min unsigned Xx to active memory Xy in MU n
BMADDn	Xx,Xy	Begin add multiple Xx to active memory Xy in MU n
BMSUBn	Xx,Xy	Begin subtract multiple Xx to active memory Xy in MU n
BMANDn	Xx,Xy	Begin and multiple Xx to active memory Xy in MU n
BMORn	Xx,Xy	Begin or multiple Xx to active memory Xy in MU n
BMMAXn	Xx,Xy	Begin max multiple Xx to active memory Xy in MU n
BMMAXUn	Xx,Xy	Begin max unsigned multiple Xx to active memory Xy in MU n
BMMINn	Xx,Xy	Begin min multiple Xx to active memory Xy in MU n
BMMINUn	Xx,Xy	Begin min unsigned multiple Xx to active memory Xy in MU n
EMADDn	Xx,Xy	End add multiple Xx to active memory Xy in MU n
EMSUBn	Xx,Xy	End subtract multiple Xx to active memory Xy in MU n
EMANDn	Xx,Xy	End and multiple Xx to active memory Xy in MU n
EMORn	Xx,Xy	End or multiple Xx to active memory Xy in MU n
EMMAXn	Xx,Xy	End max multiple Xx to active memory Xy in MU n
EMMAXUn	Xx,Xy	End max unsigned multiple Xx to active memory Xy in MU n
EMMINn	Xx,Xy	End min multiple Xx to active memory Xy in MU n
EMMINUn	Xx,Xy	End min unsigned multiple Xx to active memory Xy in MU n
BMPADDn	Xx,Xy	Begin arbitrary multiprefix add Xx to active memory Xy in MU n
BMPSUBn	Xx,Xy	Begin arbitrary multiprefix subtract Xx to active memory Xy in MU n
BMPANDn	Xx,Xy	Begin arbitrary multiprefix and Xx to active memory Xy in MU n
BMPORn	Xx,Xy	Begin arbitrary multiprefix or Xx to active memory Xy in MU n

BMPMAXn	Xx,Xy	Begin arbitrary multiprefix max Xx to active memory Xy in MU n
BMPMAXUn	Xx,Xy	Begin arbitrary multiprefix max unsigned Xx to active memory Xy in MU n
BMPMINn	Xx,Xy	Begin arbitrary multiprefix min Xx to active memory Xy in MU n
BMPMINUn	Xx,Xy	Begin arbitrary multiprefix min unsigned Xx to active memory Xy in MU n
EMPADDn	Xx,Xy	End arbitrary multiprefix add Xx to active memory Xy in MU n
EMPSUBn	Xx,Xy	End arbitrary multiprefix subtract Xx to active memory Xy in MU n
EMPANDn	Xx,Xy	End arbitrary multiprefix and Xx to active memory Xy in MU n
EMPORn	Xx,Xy	End arbitrary multiprefix or Xx to active memory Xy in MU n
EMPMAXn	Xx,Xy	End arbitrary multiprefix max Xx to active memory Xy in MU n
EMPMAXUn	Xx,Xy	End arbitrary multiprefix max unsigned Xx to active memory Xy in MU n
EMPMINn	Xx,Xy	End arbitrary multiprefix min Xx to active memory Xy in MU n
EMPMINUn	Xx,Xy	End arbitrary multiprefix min unsigned Xx to active memory Xy in MU n

Write Back subinstructions

WBn	Xx	Write Xx to register Rn.

Arithmetic and Logical Unit subinstructions

ADDn	Xx,Xy	Add Xx and Xy in ALU n
SUBn	Xx,Xy	Subtract Xy from Xx in ALU n
MULn	Xx,Xy	Multiply Xx by Xy in ALU n
MULUn	Xx,Xy	Multiply Xx by Xy in ALU n unsigned
DIVn	Xx,Xy	Divide Xx by Xy in ALU n
DIVUn	Xx,Xy	Divide Xx by Xy in ALU n unsigned
MODn	Xx,Xy	Determine Xx modulo Xy in ALU n
MODUn	Xx,Xy	Determine Xx modulo Xy in ALU n unsigned
LOGDn	Xx	Determine ROUNDDOWN(Log2 Xx) in ALU n
LOGUn	Xx	Determine ROUNDUP(Log2 Xx) in ALU n
SELn	Xx,Xy	Select Xx or Xy according to the result of previous compare operation in functional unit chain (Xx if res=1, Xy if res=0)
MAXU	Xx,Xy	Determine maximum of Xx,Xy in ALU n unsigned
MAX	Xx,Xy	Determine maximum of Xx,Xy in ALU n
MINU	Xx,Xy	Determine minimum of Xx,Xy in ALU n unsigned
MIN	Xx,Xy	Determine minimum of Xx,Xy in ALU n
SHRn	Xx,Xy	Shift right Xx by Xy in ALU n
SHLn	Xx,Xy	Shift left Xx by Xy in ALU n
SHRAn	Xx,Xy	Shift right Xx by Xy in ALU n arithmetic
RORn	Xx,Xy	Rotate right Xx by Xy in ALU n
ROLn	Xx,Xy	Rotate left Xx by Xy in ALU n
ANDn	Xx,Xy	And of Xx and Xy in ALU n
ORn	Xx,Xy	Or of Xx and Xy in ALU n
XORn	Xx,Xy	Exclusive or of Xx and Xy in ALU n
ANDNn	Xx,Xy	And not of Xx and Xy in ALU n

ORNn	Xx,Xy	Or not of Xx and Xy in ALU n
XNORn	Xx,Xy	Exclusive nor of Xx and Xy in ALU n
CSYNCn	Xx	Set up barrier synchronization group Xx in ALU n
SEQn	Xx,Xy	Set result=-1 if Xx = Xy else result=0 in ALU n
SNEn	Xx,Xy	Set result=-1 if Xx ≠ Xy else result=0 in ALU n
SLTn	Xx,Xy	Set result=-1 if Xx < Xy else result=0 in ALU n
SLEn	Xx,Xy	Set result=-1 if Xx ≤ Xy else result=0 in ALU n
SGTn	Xx,Xy	Set result=-1 if Xx > Xy else result=0 in ALU n
SGEn	Xx,Xy	Set result=-1 if Xx ≥ Xy else result=0 in ALU n
SLTUn	Xx,Xy	Set result=-1 if Xx < Xy unsigned else result=0 in ALU n
SLEUn	Xx,Xy	Set result=-1 if Xx ≤ Xy unsigned else result=0 in ALU n
SGTUn	Xx,Xy	Set result=-1 if Xx > Xy unsigned else result=0 in ALU n
SGEUn	Xx,Xy	Set result=-1 if Xx ≥ Xy unsigned else result=0 in ALU n

Immediate Operand Input subinstructions

OPn	d	Input value d into operand n

Compare Unit subinstructions

SEQ	Xx,Xy	Set IC if Xx equals Xy
SNE	Xx,Xy	Set IC if Xx not equals Xy
SLT	Xx,Xy	Set IC if Xx is less than Xy
SLE	Xx,Xy	Set IC if Xx is less than or equals Xy
SGT	Xx,Xy	Set IC if Xx is greater than Xy
SGE	Xx,Xy	Set IC if Xx is greater than or equals Xy
SLTU	Xx,Xy	Set IC if Xx is less than Xy unsigned
SLEU	Xx,Xy	Set IC if Xx is less than or equals Xy unsigned
SGTU	Xx,Xy	Set IC if Xx is greater than Xy unsigned
SGEU	Xx,Xy	Set IC if Xx is greater than or equals Xy unsigned

Sequencer subinstructions

BEQZ	Ox	Branch to Ox if IC equals zero
BNEZ	Ox	Branch to Ox if IC not equals zero
JMP	Xx	Jump to Xx
JMPL	Xx	Jump and link PC+1 to register RA
TRAP	Xx	Trap
JOIN	Xx	Join all the threads to a NUMA bunch Xx
SPLIT	Xx	Split all the current NUMA bunches back to PRAM mode threads

Parallel and Distributed Computing

Edited by Alberto Ros

ISBN 978-953-307-057-5

Hard cover, 290 pages

Publisher InTech

Published online 01, January, 2010

Published in print edition January, 2010

The 14 chapters presented in this book cover a wide variety of representative works ranging from hardware design to application development. Particularly, the topics that are addressed are programmable and reconfigurable devices and systems, dependability of GPUs (General Purpose Units), network topologies, cache coherence protocols, resource allocation, scheduling algorithms, peertopeer networks, largescale network simulation, and parallel routines and algorithms. In this way, the articles included in this book constitute an excellent reference for engineers and researchers who have particular interests in each of these topics in parallel and distributed computing.

How to reference

In order to correctly reference this scholarly work, feel free to copy and paste the following:

Martti Forsell (2010). Total Eclipse - an Efficient Architectural Realization of the Parallel Random Access Machine, Parallel and Distributed Computing, Alberto Ros (Ed.), ISBN: 978-953-307-057-5, InTech, Available from: http://www.intechopen.com/books/parallel-and-distributed-computing/total-eclipse-an-efficient-architectural-realization-of-the-parallel-random-access-machine

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen