Automatic Code-Generation Techniques for Micro-Threaded RISC Architectures

Jason McGuiness

Submitted to the University of Hertfordshire in partial fulfillment of the requirements of the degree of Master of Science by Research.

Compiler Technology and Computer Architecture Group,
Department of Computer Science Sciences,
University of Hertfordshire,
England.
July 2006

Dedicated to

the many enlightening discussions I had with Dr. Richard Harris and Dr. Andres Márquez.

Automatic Code-Generation Techniques for Micro-Threaded RISC Architectures

Jason M^{c}Guiness

Submitted to the University of Hertfordshire in partial fulfillment of the requirements of the degree of Master of Science by Research.
July 2006

Abstract

There has been an ever-widening gap between processor and memory speeds, resulting in a 'memory wall' where the time for memory accesses dominates performance. To counter this, architectures that use many very small threads that allow multiple memory accesses to occur in parallel have been under investigation. Examples of these architectures are the CARE (Compiler Aided Reorder Engine) architecture, micro-threading architectures and cellular architectures, such as the IBM Cyclops family, implementing using processors-in-memory (PIM), which is the main architecture discussed in this thesis. PIM architectures achieve high performance by increasing the bandwidth of the processor to memory communication and reducing that latency, via the use of many processors physically close to the main memory. These massively parallel architectures may have sophisticated memory models, and I contend that there is an open question regarding what may be the ideal approach to implementing parallelism, via using many threads, from the programmer's perspective. Should the implementation be at language-level such as UPC, HPF or

other language extensions, alternatively within the compiler using trace-scheduling? Or should it be at a library-level, for example OpenMP or POSIX-threads? Or perhaps within the architecture, such as designs derived from data-flow architectures? In this thesis, DIMES (the Delaware Iterative Multiprocessor Emulation System), which is being developed by CAPSL at the University of Delaware, was used as a hardware evaluation tool for such cellular architectures. As the programing example, the author chose to use a threaded Mandelbrot-set generator with a work-stealing algorithm to evaluate the DIMES *cthread* programming model. This implementation was used to identify potential problems and issues that may occur when attempting to implement massive number of very short-lived threads.

Declaration

The work in this thesis is based on research carried out at the Compiler Technology and Computer Architecture Group, University of Hertfordshire, England. No part of this thesis has been submitted elsewhere for any other degree or qualification and it is all my own work unless referenced to the contrary in the text.

Acknowledgments

My sincere thanks to Dr. Colin Egan my final supervisor, of the University of Hertfordshire, for taking on an unusual student! But also my thanks to Professor Alex Shafarenko, of the same University for his inspirational conversations, and being instrumental in my collaboration with CAPSL and the University of Delaware. Finally, my thanks to Dr. Slava Muchnick, who initiated this program of research. The following doctors also deserve honorable mention: Dr. Andres Márquez and Dr. Georg Munz. Both were inspirational conversationalists, each in their own ways...

The research presented in this thesis was, in part, supported by the Engineering and Physical Research Council (EPSRC) grant number: GR/S58492/01.

Contents

List of Figures

Chapter 1

Introduction

The memory-wall [121] is a limiting factor in CPU performance, which may be countered by introducing extra levels in the memory hierarchy [15, 121]. However, these extra levels increase the penalty associated with a miss in the memory-subsystem, due to memory-access times, limiting the ILP. Also, there may be an increase in design complexity and power consumption of the overall system. An approach to avoid this problem may be to fetch sets of instructions from different memory banks, i.e. introduce threads, which would allow an increase in ILP, in proportion to the number of executing threads. There are issues with introducing multiple threads of execution, such as they should not have data or control flow that is inter-dependant between any of the currently executing threads. Another issue is that the cost for creating, synchronising and destroying threads should be very cheap, which constrains the architectural design. The reason for this latter constraint is that the latencies to be mitigated against would be pipeline stalls, usually very short periods of time, potentially between a few to tens of clock cycles. Such short threads, that are designed to mitigate against pipeline stalls, this thesis shall term as *micro-threads*. This definition is in slight contrast to the definitions used within [13, 68], where the motivation for the definition came from the differing size of the thread, i.e. that they lacked a stack and context. Given that the threads to which this thesis refers, and the threads of [13, 68] are all used to maintain pipeline throughput, then this modified definition has some justification. Note that these micro-threads are not operating-system level threads, which have large context, are potentially pre-

emptible and used for process-level parallelism. Micro-threads would be designed to have very little context, making creation and destruction cheaper.

Various architectures have been proposed that could support micro-threads:

- The architecture of [13, 68], which is designed to support the smallest variant of micro-threads.
- The CARE (Compiler Aided Reorder Engine) architecture described in [75], that supports *strands,* a variant of micro-threads, that fulfil the same goal, thus come under the definition of micro-threads used in this thesis.
- The integration of processing logic and memory [21, 41, 105, 106] within the same chip, termed PIM. Such integration may also improve both data-processing and data-access time.

In this thesis the application of micro-threading to the final PIM architecture is what will be examined in most detail, with occasional references to the other architectures.

A problem with integrating processors and memory in the same space is that the processor speed and the amount of memory are reduced [21]. This may be overcome by connecting multiple, independent PIM cells, where the resultant architecture is described as cellular. In this multi-threaded organisation, every thread unit serves as an independent single-issue, in-order processor, thus able to potentially access memory independently, depending upon the exact details of the architectural design.

This gives rise to a number of code-generation problems, some of which are discussed in appendix C, centred around the fact that to provide computational power, these systems are massively parallel. It is common folklore in the programming community that writing correct and efficient multi-threaded programs is hard. This problem could be compounded for such cellular architectures. Thus, considerable research effort has been targeted at code generation, including thread generation, to support such hardware. There is likely to be much research to do: to develop compilers to generate multi-threaded code, create lower-level libraries that ease the burden of creating such code, and write debuggers that allow the programmer to effectively debug such programs.

Thread-generating compilers exist; for example, HPF and UPC [45]. The Fortran-based HPF is very useful for mathematical problems, but less so for other problem domains. Both compilers specialise on parallelising loop-constructs. Other C and C++ parallelising compilers exist, but are largely based upon the OpenMP library, for example IBM XL Fortran and Visual Age C/C++, which also tend to focus upon loops and a source of parallelism. Alternatively, higher-level approaches, such as a compiler that may automatically create threads using the *split-phase constraint* exist for such architectures as EARTH [109]. The *split-phase constraint* may be loosely defined as when the compiler may generate a synchronization variable, and a destination thread for a potentially, long-latency load from remote memory. This EARTH compiler attempts to fulfil the promise of thread-generation for the programmer: it is automated, general-purpose - not limited to loops - and the schedules it creates are provably fast and correct.

Moreover, the different memory hierarchies within cellular architectures add to the multi-threaded code-generation problem. Research is in progress to address this problem: for example by placing hardware memory-banks that have different access and consistency models at different address ranges in the memory-map of the virtual machine, known as *location consistency* [40] is one approach. The EARTH compiler and UPC both provide language, hence compiler support, for such features using the split-phase constraint, or the use of the *strict* and *relaxed* keywords, respectively.

The library-based approach to threading has often been made less effective by a lack of language support, that would aid the expressiveness and use of thread-related constructs (for example threads themselves and synchronization mechanisms). For example, the use of pragmas in the various implementations of OpenMP, and the fact that general-purpose languages have been very slow to adopt a sufficiently sophisticated abstraction of the features of any machine model. C/C++ has had the *volatile* keyword for over a decade, but has made very limited use of it in supporting shared data, that may be accessible by more than one thread, an obvious use of the keyword. (Indeed this use is to be introduced into the next C++ standard, to be finalised not before 2009.) This limitation has been noted (at the Association of C and C++ Users Conference, 2005, by B. Stroustrup, in one of his keynote presentations,

and by others) and has apparently hampered development of multi-threaded programs and the development of compilers that might automatically generate threads. The author contends that library-based solutions to threading are too dependant upon the programmer to use correctly. For example, the explicit use of locks in programs is prone to error, with deadlocks and race-conditions that are hard to track down are easily introduced.

The development of suitable tools to debug multi-threaded applications has been slow. Some tools are available (*strace, truss, pstack* and various debugger) but are very limited in functionality, with regards to threading. More useful debuggers are in development, for example for Cyclops [29]. But these are few, with currently limited functionality. Further development in this area would be vital to allow the programmer to debug their code on such systems. A more important aspect of these tools would be to aid the programmer with regards to reasoning about the function of their multi-threaded code, and thus avoid such bugs.

In the author's opinion, the leaders in this field are aiming at a language, not library, based solution, which would be the appropriate level of abstraction for the expression of parallelism within a program. The compiler support would allow the development of more powerful multi-threading abstractions, such as various algorithms, that would help to divorce the programmer from the complex details of the underlying architectural support. But there are limitations in the direction of such current compiler developments, for example, UPC apparently exposes only loop-based parallelism and HPF requires explicit statements within the code to make the compiler generate multi-threaded code which also directed towards parallelising loops. The author contends that this would be far too limited for application to general-purpose programs.

As identifying parallelism both correctly and efficiently has been very hard for the programmer to do, the author contends that they should not do it. When such massively-parallel architectures are developed, this process should include time to develop libraries that plug into the target compilers to allow them to generate efficient code for that architecture. Thus the programmer would identify variables and functions that they believe they may be able to parallelise, to guide the compiler.

The compiler, equipped via these libraries with a detailed machine-model would be able to refine and hone these gross indications in the program to generate efficient code. The author experienced only limited effort investigating the software aspect of the code generation problem for massively parallel architectures. Unfortunately, if this case should continue, this shortcoming could adversely affect the popularity of such systems and maintain the perception that massively parallel architectures are too specialized and thus too expensive to be of more general use. Given the popularity of multi-core processors, this position is set to become even more untenable.

Chapter 2

Related Work

2.1 The VLIW Origins

The research that has been done in the field of multi-threaded architectures, may be considered to have been heavily influenced by the work on VLIW architectures: one can consider them to have a limited number of "live" threads at any one instant, limited not only by the number of slots in the instruction word, but also by the ability of the compiler to identify such instruction-level parallelism. Some research work [83, 117] demonstrated that, in the SPEC95 benchmark suite, there has been potential for a large number of independent threads, up to the order of thousands.

Unfortunately this motivating result was for VLIW machine-models with certain, ideal parameters; a common limit has been the number of available registers, or bypass buses, or an oracle branch predictor within the compiler. This gave impetus to the architecture field to research these rich topics, and has provided very effective *dynamic*, rather than compile-time branch predictors. But the VLIW compilers, the trace compilers of the time, required a compile-time branch predictor to produce code that did not need expensive recovery mechanisms, and enable the compiler to perform the whole-program, code-motion optimizations it needed to do to extract the ILP from the programs. Results for the register problem have been similarly mixed: due to the multi-ported nature of the register banks, there is a physical and technological limit: having more registers scales the area of the chip linearly, but more register ports (for bypass buses) scales the area geometrically. Technological

limitations in chip fabrication limit the yield of the chips: the larger the area, the lower the yield in direct proportion. Thus adding a sufficient number of register ports will always reach a limit in the current technological ability to produce economic quantities of such chips.

The instruction density in VLIW code decreased for various reasons:

- a lack of an effective compile-time branch predictor,
- combined with limited register resources,
- true data-dependencies,
- and structural hazards

all of which meant is was necessary to inject no-ops into the instruction stream. These no-ops have been of vital significance: they were a direct indication of the inefficiency of the compiler and tool-chain, hence architecture, to extract ILP from the instruction stream, and indicate an inefficiency of both the software and the compiler. Consequently the effectiveness of the VLIW architecture as a technique to increase performance, via extracting ILP, by re-compiling the source code had been constrained.

2.2 Beyond VLIW: Super-scalar: the combination of branch predictors, speculation and memory hierarchies

But the research yielded very useful results: the development of dynamic, as opposed to the compile-time branch predictors. These meant that speculative execution of code was much less likely to be wasted work. Thus the advent of super-scalar processors, but these had their problems: performance was hindered by slow memory speeds. So small caches were implemented, based upon the assumption of data and control locality. The size of a hardware cache has been chosen to be roughly 10% of the average size of the executing program the related data. These caches have

been composed of very high speed memory, which has been costly to implement. They were also placed directly in line with the IF stage of the pipeline, allowing very high-speed instruction-fetch from the cache, if there was a cache hit [80]. Also, regarding instruction fetch: the accuracy of the branch predictor and placing it very early in the pipeline has been vital. This is to allow the branch target addresses to be obtained (potentially via the BTC, or via a default prediction, or a dynamic predictor may be used) before the instructions that would generate the result of that branch condition. This allowed the instruction cache to pre-fetch cache-line sized amounts of instructions from slower hierarchies, using the pre-computed, predicted, branch-target address, and deliver them with minimal pipeline stalls to the IF stage. The data cache has been more complex, but the concept of implementing a small, write-back, high-speed amount of memory so that register writes would be directed to this memory has been relatively simple: it would act as a buffer to the lower-level, slower memories, and allow memory reads to be potentially serviced directly by the data cache instead of from the lower-level memory hierarchies. Another major factor has been out-of-order instruction execution: if there were sufficient processor resources, instructions could be executed in parallel, although they would be fetched in-order and potentially retired out-of-order. Moreover instructions that completed faster need not be held up by slower instructions that were ahead of them in the instruction stream. The use of a scoreboard or register file [59, 80] allowed the data-dependencies between registers to be dynamically computed whilst the instructions were in flight in the pipeline. When these caches were combined with branch prediction and speculation even more ILP, and performance, could be extracted from the input instruction stream. In these architectures, the retirement of instructions was linked to an architectural state (potentially implemented via a reorder buffer) that, if a mis-prediction occurred, would have to be rolled back, and the instruction fetch re-started from the alternative branch. Also, if a processor were to implement precise interrupts, for example to implement processor exceptions, then a similar roll-back, or completion, of in-flight instructions would need to occur to ensure that the processor would be in an architectural state that would be consistent with the sequential program state.

2.3 Parallel Architectures

The roll-back implicitly implemented within super-scalar architectures has been viewed as a problem: the increased state due to deeper pipelines makes the chips much more complex. This increasing complexity has been viewed as one of the limits to the scalability of the super-scalar architecture. The implicit assumption in the von Neumann architecture underlies this design, therefore more radical alternatives would need to be researched if increased performance may be obtained under such constraints, for example data-flow based compilers [12, 99] and computer systems [48, 57]. But the data-flow architecture itself had problems: the architectural state was reflected in increasingly many registers, with increasingly many ports, thus complicating chip design, in a similar manner to the VLIW register problems.

The implementation of large quantities of memory with mixed execution units may be seen to have led to a few avenues of research. The ones that are pertinent to this thesis are:

- EARTH and CARE,
- the micro-threaded architecture,
- and cellular architectures such as IBM BlueGene/C and Cyclops.

In general these architectures examine various techniques by which the excess performance of the execution units may be used to ameliorate the relatively limited instruction and data throughput rate from the memory subsystems. Threading the program attempts to divide the sequential program into data and control dependent threads. These dependencies imply a partial execution order upon the threads that must be satisfied to maintain the consistency of the original program, as expressed by the programmer in the target language, which has often been a sequential language. By this technique the von Neumann architectural concept of strict instruction fetch-decode-execute-writeback could be avoided. Instead there could be, effectively multiple execution units, each executing as a von Neumann architecture, within a whole architecture that would be applied to the program as a whole, thus attempting to mine such ILP as may be available within that program.

2.3.1 EARTH, the EARTH compiler and CARE

2.3.1.1 The EARTH architecture

The EARTH architecture [53], was composed of: a synchronization processor and an execution processor, linked by two queues. The program would be written in Threaded-C, such that those threads within the program would be scheduled by an synchronization unit to execute on connected execution unit, but only if all of the related dependencies had been satisfied. Due to the multi-processor nature of the architecture the thread size would be chosen to optimize execution so that any reduction in efficiency due to long latency delays caused by inter-processor communication would be minimized. These delays could be of many orders of magnitude longer than latencies due to branch mis-predictions, or local memory accesses. Threaded-C required the programmer to annotate their program with thread constructors to direct the compiler to generate multi-threaded code.

2.3.1.2 The EARTH Compiler

To overcome the necessity for the programmer to annotate the program, Tang in his work [109], describes a compiler that was able to take a C program and suitably annotate it with the appropriate threads. Most importantly this could be done without the programmer's intervention.

The technique described in [109] is as follows: the compiler tried to identify, with the potential aid of type modifiers, those operations that may have caused long latencies. Those memory accesses would be labeled using the *local* or *remote* type modifier, and if no modifier were used the compiler had to assume that the access was remote, therefore the type modifier would be *remote*. The remote type modifier indicated to the compiler that the memory access would be of long latency. These long-latency operations, for example, memory accesses or function calls, would then be split into two threads. The first thread was the original thread and the second thread contained the code that was data-dependent upon the long-latency operation. To ensure that the data dependence was satisfied a synchronization variable was introduced, such that the second thread waited upon this synchronization ob-

ject before it could execute, which [109] terms as the *split-phase* constraint. To generate these threads the compiler created a data dependence graph of the input program, with the edges in the graph being labeled as remote and local. Those remote edges would be split by the compiler using the split-phase constraint. The compiler also builds up a *program dependence flow graph* in which the data and control dependencies of the program were hierarchically captured. This graph included the threaded representation of the original program from which the compiler then identified an optimal order that satisfied all of the constraints. This graph also allowed the compiler to identify further optimizations:

- To reduce thread switching costs, control and data independent threads should be merged. This was done by computing the remote level of each node, and merging those that have the same remote level.
- Within a thread, registers should be re-used and data shared with other instructions within the thread, to enhance locality and sequential performance of the instruction stream.
- Long latency operations could be covered by control and data independent local operations, providing that the overall control and data dependencies are satisfied.

In [109], Tang showed that the optimization problem posed by combining the above details and minimizing the total execution time was NP-hard. Thus an alternative partitioning algorithm was required, to minimized compilation time. Tang showed that the list-based scheduling algorithm selected was no worse than twice as slow as an optimal schedule of the nodes. This bound may be improved upon by reducing the cost of remote communication. Tang also examined the use of the various heap based analysis to aid the thread partitioner so that it can create more threads, if required.

The results presented in [109] showed that for randomly generated program graphs, the list-based, thread-scheduling algorithm produced code that was within 7% of the ideal run-time, which was close to an optimal schedule. Also, for the

custom benchmarks used by the paper, the thread scheduler produces code that was comparable in performance to optimized, hand-written code. Their results showed that the heap analysis technique improved the performance of the scheduler, which made use of the heap analysis to optimize the thread performance.

2.3.1.3 The CARE Architecture

In [75] the basis of the large threads implemented within the EARTH was re-examined. In this case the threads were much reduced in size. The concept behind CARE was that the instruction fetcher within the pipeline required more guidance to be able to fetch instruction pointers to single-entry single-exit basic blocks, termed *strands*, that could be executed without stalls within the pipeline. Therefore during execution, the instruction fetcher would have an opportunity to identify other such strands for subsequent execution. Indeed each strand would have a set of associated *firing rules* that, if satisfied, would allow that strand to be scheduled for subsequent, stall-free, execution. These firing rules would represent the data and control dependencies upon which the instructions within the strand depend. Thus the instruction fetch unit would contain a set of strands that have all of their firing rules satisfied, ready to be executed, and another set of strands, which are awaiting their firing rules to be satisfied. The compiler, in this architecture, would create the strands, and identify the firing rules and populate that data structure. Moreover, the initial ordering of the strands within the instruction stream would be performed by the compiler. But the architecture, at run-time would be allowed to re-order strands, if their firing rules were satisfied.

2.3.2 The Micro-Threaded Architecture

In [13] a mathematical model was presented that examined the latencies from a generalized memory unit, modeled as a queue, to a generalized processing unit, i.e. requests for data. Their results for a local memory system, as opposed to networked, are reproduced in figure 2.1. They demonstrated that to obtain over 80% performance there need only be over 4 threads ready for execution at any one instant in the program. This result was *independent* of the type of input program.

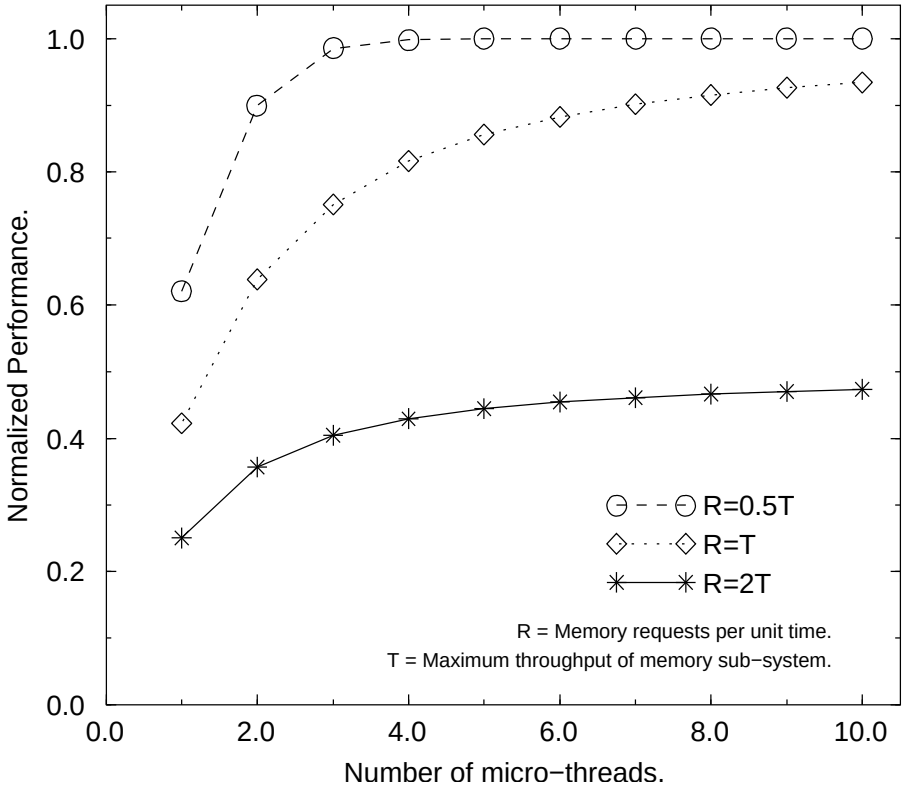


Figure 2.1: $P(n)$ for conventional memory with $L_0 = 1/T_0$, taken from [13].

It was also independent of the exact memory sub-system implementation. Indeed the only assumption that was made was the fact that the processor architecture can support micro-threading, the exact implementation of the micro-threading being abstracted out of the model. From the studies of available ILP within general programs, it would seem that the implementation of the technique of micro-threading in a processor would be extremely effective in maintaining processor throughput during memory loads. An important property of the micro-threads described in [13] was that the cost of thread creation, destruction and synchronization must be very cheap, due to the number and frequent switching of micro-threads. This property of micro-threads implied that there must be efficient hardware support for them. To transform a generic program into a micro-threaded program implied that the control constraints within the sequential program must be transformed into thread creation and synchronization constraints. This task would be achieved by a micro-threading stage within a suitable compiler. Further work [68] within this field has demonstrated the feasibility of such an architecture. A simple schematic of their

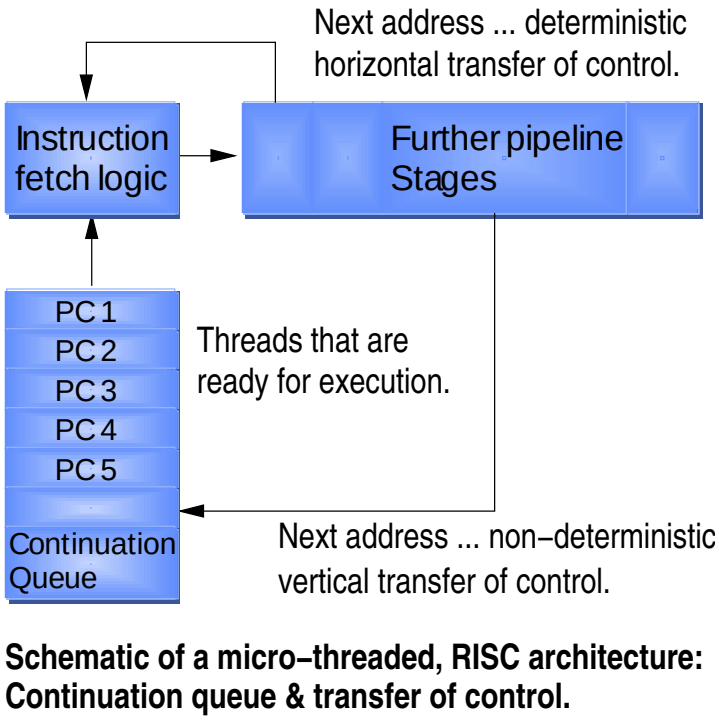


Figure 2.2: Schematic of a micro-threaded, RISC architecture.

implementation is provided in figure 2.2.

In this architecture, there are many very short threads, perhaps only 2-5 instructions in length. They wait upon only one data item, that may be viewed as a simplified version of the firing rules of CARE. The PCs of those threads that are ready to execute are stored in a continuation queue, for eventual execution within the pipeline. Because of the architectural speed of thread creation, synchronization and destruction, no speculation would be done: all of those features would be converted into micro-threads, thus the execution pipeline could be a relatively simple RISC-like pipeline.

2.3.3 IBM BlueGene/C and Cyclops

This architecture will be discussed in much more detail in chapter 3.3 of this thesis, but for the purposes of this section, a brief summary will suffice. This architecture was a PIM-like architecture, termed cellular, that implements a number of execution units and memory units on one die. Thus it has the ability to execute many threads, has fast memory access, and may be viewed, in some sense, as between the EARTH

architecture and the micro-thread architecture, in terms of a threading model.

In order to overcome the von Neumann-derived memory wall, some method of overcoming the implicit data-fetch delay should be implemented within the architecture. Moreover, such implementations usually imply multiple threads of execution, which further implies data and control dependencies that must be resolved, either at compile or run-time:

- Within EARTH and CARE this is at compile-time: the synchronization unit has explicit dependencies upon which it must wait, which have been generated at compile time.
- Within micro-threaded architectures, these dependencies may be left to be resolved at run-time, as long as all potentially data-dependent instructions are suitably annotated by the compiler.
- Within Cyclops, as will be presented in chapter 3.3, the control and data dependencies are much more complex due to the increased complexity of the architecture and the massive parallelism it makes available.

Eventually this implies that some technique must be used, either explicitly or implicitly by the programmer to generate the required threads for the architecture.

During the research program, I chose to concentrate upon the Cyclops architecture for the rest of the program, as an example of the problems with programming for such sets of threaded architectures.

Chapter 3

The limitations of super-scalar architectures: the memory wall

The combination of data and instruction caches effectively decouples the processor from the speed of the main-memory, by simply introducing more layers of caches in the memory hierarchy. This decoupling has been highly successful: the increase in performance of processors of the past decades has been greatly influenced by the dramatic increase in clock speed. The original 8086 was clocked at roughly 4MHz with no instruction cache, the latest Pentium 4s have been clocked at over 3.4GHz [56]. These latest Pentiums could retire instructions at a rate of roughly ten-times the main memory speed by using two to three cache levels. But to get such high speeds the pipeline depth has had to be increased. The Pentium 4 has over 20 stages; the AMD Opteron has 10-12 stages, and has been clocked at approximately 2.6GHz. With these processors, if a branch mis-prediction or processor exception should occur and the state would have to be rolled back, then instruction fetch and the pipeline must be restarted, so it would take increasingly long in a 20 stage pipeline to begin retiring instructions after the restart. The accuracy of the branch predictor has been paramount, to avoid such time-consuming re-starts. But if the processor speed were to increase, then more stages may be needed, and branch-mispredictions would become even more costly. Moreover, the increased latency of instruction fetch from the mis-predicted branch would increase due to the divergent relative speeds of the processor and main memory. This problem has been termed

the *memory wall* [117].

3.1 Multiple cores and massively parallel architectures

The problem of the memory wall may be viewed as an effect of the relative performance difference of main memory to processor speed. If the instruction throughput could be increased by reading instructions from different memory banks, then the instruction issue rate is potentially limited by the number of available memory banks, and IF stages attached to them.

Multi-core processors develop this idea. Let us suppose that the OS supports preemptive multi-tasking, and these OS-level threads are guaranteed to have the inter-thread, data-dependencies explicitly specified using kernel-level (thus architectural) synchronization primitives. If the resources used for developing higher clock speeds were instead used in implementing another core within the processor package, this extra core would be viewed as an extra processor by the OS for scheduling threads upon. Moreover, if the program were suitably written, it could take advantage of any extra processor resources. But this requires extensive and potentially difficult modifications to the source code to allow it to take advantage of such extra resources. Moreover, the use of OS-level threads is expensive: they have a lot of context, because each thread must not only retain the processor state, but the OS state, if it were to be context-switched off the processor. Architectural-level threading would seem to be a faster and more simple approach. Another limitation with multiple processor cores is that the processor cores take die space away from the caches and branch-predictors, that are proven, high-performance solutions.

Furthermore, there are costs associated with switching between an OS-level thread with considerable context. These costs include the memory access times to flush the OS and architectural states to main memory, and the instruction- and data-cache misses inherent in such a context switch. A technique to avoid these latencies may be to reduce the thread context to a level such that any such context could be maintained in the processor, without having to be flushed to a lower mem-

ory hierarchy. But this implies a dramatic reduction in context: for micro-threading the context has been limited to only a program counter - an extreme example. Moreover this reduction also implies that these threads would be unlikely to be managed at the OS-level.

To counter the memory latencies inherent in the super-scalar designs, the approach of placing the execution pipelines as close to the memory as possible may be taken. In this context *close* means that the memory and the execution pipelines are on the same die. Each instruction fetch stage and the data-bus of a pipeline would be fed directly from an independent bank of memory. Thus the instruction fetches and, more importantly, data reads and writes can occur independently of other pipelines on the same die and other dies. This integration has the advantage that the latency of memory access would be dramatically reduced. But to allow such integration, the pipelines are usually much more simple than a super-scalar pipeline. Often they have no branch prediction, thus no speculation, which allows the space that a pipeline consumes on the die to be dramatically reduced, thus allowing more memory and more execution units per die. For example, in the picoChip [33] design there are approximately 308 VLIW cores and a similar number of DSP pipelines on one die, with each VLIW core having direct, 1-clock cycle access to approximately 64K of RAM. Alternatively in the IBM BlueGene/C design [4], described in section 3.3, more sophisticated 64-bit cores are implemented with approximately 64K of software-controlled data cache, and another 4Gb of RAM on chip, but with a reduced number of pipelines, in this case approximately 96. Such chips offer a considerable instruction retirement throughput. To further increase the bandwidth, the picoChip has 4 ports implemented on it for accessing other picoChips in a grid arrangement, and a memory port for accessing off-chip memory. To date, arrays of up to 16 picoChips have been built. The IBM BlueGene/C design has 6 inter-chip connection ports, allowing a cubic array of chips. Such an arrangement of IBM BlueGene/C chips has been termed as a cellular architecture: each cell would be an IBM BlueGene/C chip. The size of the entire IBM BlueGene/C array has been envisaged to scale up to potentially 10,000,000 individual cells.

3.2 The programming models: from compilers to libraries

With such compute bandwidth, and parallelism, a number of problems for the programmer have been raised, primarily these are focused on the problems of memory reads and writes. Super-scalar chips have had mechanisms to hide these problems from the programmer, but the cellular chips such as picoChip and IBM BlueGene/C do not. Thus the programmer needs to know how memory reads and writes interact with:

- the software-controlled data-cache attached to that pipeline,
- the software-controlled data-cache of other on-chip pipelines,
- any global on-chip memory,
- the software controlled data-caches of other off-chip pipelines,
- the global on-chip memory that is on any other chips,
- any global memory that is not on any chip
- and finally, given the massive parallelism available, how to make efficient use of it.

These issues give rise to various programming models, but initially the last point will be discussed. Given the evidence of ILP studies, the efficient use of the massive parallelism for general purpose programs such as SPEC2000 is highly unlikely to be able to be parallelized to the extent of using a fraction of the resources of the IBM BlueGene/C design, and similarly with the smaller picoChip. The answer would be that these architectures eschew the aspiration of being practical for general-purpose use. Instead they target specific, embarrassingly-parallel problem domains.

For a programmer, the memory access models are important to understand, or to have a library or compiler that hides the details from the applications programmer. In the remainder of this thesis the author will focus on the IBM BlueGene/C architecture, and a prototype implementation of it called Cyclops, that was implemented

at CAPSL at the University of Delaware in collaboration with the University of Hertfordshire. In the following sections the memory access models will be discussed, leading on to a presentation of the author's experience in developing a program for such an architecture. The experience gained from this will allow the author to discuss the major problems that were faced, how, if at all, they were overcome, and the outstanding problem domains that, in the author's experience, would hinder the acceptance of multi-core chips and, moreover such massively parallel designs as IBM BlueGene/C.

3.3 IBM BlueGene/C, Cyclops and DIMES/P: the implementation of a cellular architecture

The IBM BlueGene/C architecture is described in detail in [4]. Briefly, this architecture consists of a large number of thread units, an equal number of memory banks and a large crossbar on one die. The execution thread-units are linked to each other and the memory banks via the crossbar, which also has at least 8 off-chip interconnects. These interconnects may be used to connect more of these chips together in a large 3-d mesh. Of the order of 160 thread units are on a single die, with the order of 2-4 Gbytes of DRAM, on-chip. This means that per chip there is a large amount of available parallelism, and considering that the 3-d mesh may contain of the order of 100,000 of such chips. A further factor in this design is that there is no data cache: instead there is a specialized portion of each DRAM bank that is directly accessible via a related thread unit. Such a portion of the DRAM is termed the *scratch-pad memory*, and is effectively a software controlled data cache. This scratch-pad memory is accessible from that related thread unit without having to access the crossbar. The other memory, not associated with any particular thread-unit is termed as *on-chip memory*. This gives rise to different memory access models. These memory access models are related to the work on location-consistency, described in [40, 124]. In brief, this is the concept that if a set of memory locations are accessed from two different thread units, the thread units will experience different memory access models of those memory locations, upon simultaneous access.

For example: simultaneous accesses, by different thread units, to location 1 might provide program consistency as the memory access model, whereas for location 2, with simultaneous accesses, by different thread units, this might provide sequential consistency. With regards to IBM BlueGene/C the scratch-pad memory only guarantees program consistency with regards to memory accesses. But for any memory accessed via the crossbar, it guarantees sequential consistency.

At CAPSL much work had been done in collaboration with IBM with regards to an implementation of the BlueGene/C architecture called Cyclops. Initially, this work was implementing CyclopsE [21], which was developed into Cyclops64, [30]. The CyclopsE architecture was prototyped in hardware, called DIMES/P, [90,91]. DIMES/P was used as the platform for executing the programming example, described in section 4.

With regards to any later discussions, it is very important to remember that each of these architectures, IBM BlueGene/C, Cyclops64, CyclopsE and DIMES/P display the same features: multiple thread units and multiple memory consistency models. This is simply because they are all implementations of these same underlying concepts.

3.4 Programming Models on Cellular Architectures

The hardware differences between cellular and super-scalar architectures indicate that different programming models, are required to make effective use of the cellular architectures [40, 41, 120]. In the first two of those three papers, their author propose the use of a combination of execution models and memory models, as already described in sections 3.2 and 3.3.

The primary concerns when programming DIMES/P, and thus any Cyclops-based architecture, were:

- How to manage the potentially large numbers of threads.
- How to easily express any parallelism within the input source-code.

- How to make correct, and most effective use, of the memory consistency models.

Some research has already been done regarding programming models for the threading, such as using thread percolation as a technique to perform dynamic load-balancing [18, 53, 61]. Another piece of research [22] investigated using multi-level scheduling-schemes: a work-stealing algorithm at the higher-level and a multi-threading technique at the lower-level to hide communication latencies. A further piece of research [37] investigated the use of filaments as lightweight threads to efficiently implement thread control.

3.5 Programming for Cyclops

Cyclops has a set of particular concerns associated with programming for it, some of which have been investigated, but for alternative architectures. For Cyclops, a reasonable technique for implementing memory consistency models, thread management, and finally making use of any parallelism was investigated.

This started with investigating how to easily implement the memory-consistency models. This was relatively simple: earlier, unpublished, work on the GCC-based compiler had implemented a simple algorithm: all static variables were stored in on-chip memory, and the function call stack, including all automatic variables was placed in the scratch-pad memory.

As there was no language-level support for thread management, a library had to be implemented to support the thread management instructions in the Cyclops ISA. An early version of TNT [29, 31], called *cthreads* was used as the basis for creating a higher-level C++ abstraction. The author considered that the cthread implementation, that closely followed a POSIX-Threads API, was far too primitive to be effectively used for programming Cyclops. The simple C++ API that was developed also included thread-management, critical-sections, mutexes and event objects to allow for easier management of the lower-level objects.

An abstraction of the extraction of parallelism from the range of possible example programs was not implemented for this thesis, as this was considered to be

potentially too closely coupled to the actual program in question. In the author's opinion, not performing this abstraction of parallelism was flawed, because it is where the crucial, further generalisations take place that allow a programmer to implement an algorithm with far less regard for the underlying architectural features. Thus the programmer would obtain much greater benefits from this more powerful abstraction.

To test these ideas, and the Cyclops architecture, a simple program was chosen. It had the properties that it was a small problem and embarrassingly parallel, ideally suited to CyclopsE. Thus an implementation of a program to generate Mandelbrot sets was created, which will be described in the following chapter, 4.

Chapter 4

Programming the Mandelbrot Set Algorithm for Cyclops

In this chapter, which is a more detailed description of the work done in appendix C, the salient details of the Mandelbrot set and an informal algorithm will be given for generating the set. How this algorithm may be multi-threaded is presented, with particular attention to the implementation used for DIMES/P [90]. This is a prototype of the DIMES hardware that implements a reduced version of CyclopsE [21]. Alternative algorithms are also presented, but were not implemented. A description of how the threaded algorithm was implemented on the DIMES/P platform will be presented, followed by an example of the application running and the operation of the work-stealing algorithm.

Further details and the various presentations which were based upon this work are given in appendices B (this was a presentation give to the University of Hertfordshire, upon my return from CAPSL, introducing DIMES and my work done there), A (this was a draft paper prepared at CAPSL in collaboration with Dr. Egan, for submission to various conferences) and C (this was a conference paper that has been accepted for publication at ACSAC06).

4.1 An Introduction to the Mandelbrot Set

The Mandelbrot [10, 72] set is intimately related to the Julia set[1] [60], discovered in the 1910s. They are both mathematical entities called *fractals* relating to the fact that they have a non-integer dimension. Fractals are part of the branch of mathematical called *Chaos Theory*, which may be defined as the term for those theories relating to pseudo-random mappings and functions. The applications of Chaos Theory is widely varied and includes such applications as compression [85], cryptography [32], economics [82], seismology [114], the shape of naturally occurring objects [10] such as clouds, trees [6] and landscapes, medicine such as the modelling of fibrillation in the human heart [44], which is apart from the pure mathematical or aesthetic nature of the objects.

Both the Mandelbrot and Julia sets may be created by iteration of a very simple equation:

$$z_{n+1} = z_n^2 + c \tag{4.1}$$

In this equation, z_n is a complex number, where $z_0 = 0$. c is also a complex number, which is initialised to a value constant throughout the iterations. The iteration of equation 4.1 terminates when:

1. Either n reaches the so-called "maximum iteration" value, m, a fixed constant, greater than zero.

2. Or $\mid z_n \mid$ exceeds the so-called "bailout" value of 2, usually set to the real value $4 (= \mid z_n \mid^2)$, for efficiency reasons. It has been proven that $\mid z_n \mid \rightarrow \infty$ once $\mid z_n \mid \geq 2$.

To generate the Mandelbrot set, algorithm 1 is used.

Usually the selection of c is not random, but a "raster-scan" of the complex plane. It is not necessary to scan the whole of the complex plane, as a property

[1] Each point in the Mandelbrot set is an "index" into the Julia set for that point.

Algorithm 1 The classic algorithm used to generate the Mandelbrot set.

1. Set the value of m, the maximum iterations, greater than zero.
2. Select a point from the complex plane, and set c to that value.
3. Initialise $n = 0$, $z_0 = 0$.
4. Execute equation 4.1.
5. Increment n.
6. If $\mid z_n \mid^2 \geq 4$ then that c is not in the set of points which comprise the Mandelbrot set. Go to 2.
7. If $n > m$ then that c is in the Mandelbrot set, i.e. $c \subset M$. Go to 2.
8. Go to 4.

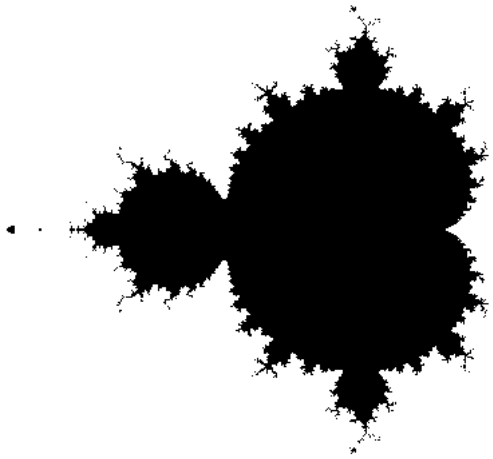


Figure 4.1: The classic Mandelbrot set image generated by "Fractint" [119]. Points coloured black are in M.

of the Mandelbrot set is that it is entirely contained within the circle of radius 2, centred on the origin of the complex plane. Another important property of the conversion to floating-point arithmetic is that the distance between the successively selected points c is a finite number representable by a floating-point number, and non-zero. In other terms, this distance is the resolution at which the set is created.

Usually the set of points M is displayed as an image, with those points in the set coloured to contrast with those that are not in the set. This gives the classic image in figure 4.1. The black region in figure 4.1 is a basin of stability of algorithm 1. Those points of which it comprises remain within a finite distance of the origin,

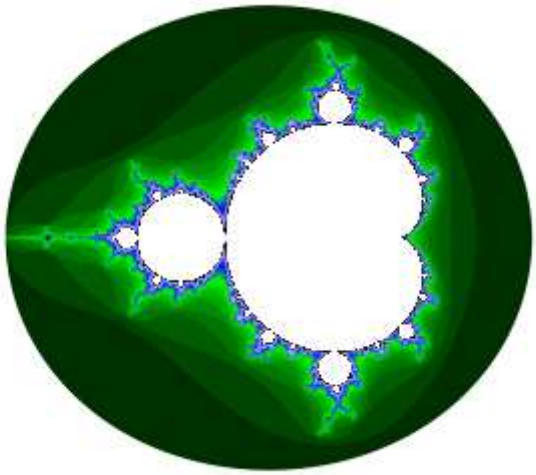


Figure 4.2: A false-colour image of the Mandelbrot set generated by "Aleph One" [71].

i.e. $\mid z_n \mid < \infty$. Those outside this region are unstable, and eventually $\mid z_n \mid \rightarrow \infty$.

More commonly, the points c have a value assigned to them that is derived from n, the iteration at which algorithm 1 terminated for that point. This gives a false-colour image, as shown in figure 4.2 , in which the points c of similar colour are termed "level-sets", basins of stability identified by the algorithm that enclose the Mandelbrot set.

4.2 Threading and the Mandelbrot Set

An important property of algorithm 1 to generate the Mandelbrot set is that the classification of each c in the complex plane is independent of the classification of any other c. Therefore the Mandelbrot set may be implemented as a massively parallel application, thus potentially suited to cellular architectures. Studies of alternative implementations for different architectures, such as fine-grain threaded-architectures [37] and NUMA architectures [22] have already been done. For cellular architectures, another important feature of this classification process is that the floating point support required may be implemented in fixed-point arithmetic using up to 32 bits for the digits, as DIMES/P [90] lacks floating point support.

The Mandelbrot set may be implemented using one algorithm per thread unit

within the cellular-architecture machine. This approach would work well for massive clusters of cellular computing nodes. (Remember that for an image of 100×100 points, c, 10,000 thread units would be required with this technique.) Moreover, the classification of any randomly selected c may take between 1 and m iterations of the algorithm. In general it is not possible to know in advance how long such a c will take to classify. Therefore the computation time would take approximately m times the time per iteration loop in algorithm 1.

Due to the properties of DIMES/P [90], this technique was not possible, as there were only 8 thread units between two processors. The chosen implementation, derived from the implementation used in [71], had the complex plane divided into a series of horizontal strips. Separate *render threads*, as the classification of the points c within each strip is independent of such classification on other render threads. Therefore each render thread implements a slightly modified version of algorithm 1, which is provided in algorithm 4. Only the coordinates for the bounding rectangle are inter-related between the render threads. However, each strip will, in general, take a different amount of time to render, thus the render threads will complete their assigned portion of work at different times. This lead to the addition of a load-balancing algorithm moving uncompleted work to threads that have already completed their assigned work. Thus a work-stealing algorithm 5 was added to perform the load-balancing between the render threads. Alternative implementations of the Mandelbrot set using a work-stealing algorithm [22] or fine-grain threaded algorithm [37] exist.

The updates to the start, x, and finish points of the strips for the render threads T_c and T_l are performed atomically - the threads are suspended whilst these updates are done, either because T_c is stopped or because T_l is stopped by using a mutex. (A mutex is required as the data to be updated is a two complex numbers, x and the finish point, these must both be updated as a pair, atomically. In this implementation a complex number consists of two words - one for the real part, one for the imaginary part.) This is a dynamic-programming solution to the load-balancing problem of work distribution between the render threads. Moreover, the algorithm is robust: if the estimated completion-time, t, has an error, which it is

Algorithm 2 The render-thread algorithm.

1. Set the value of m, the maximum iterations, greater than zero. Set the estimated completion-time, t, to the largest, finite, representable time-period possible.
2. Set $c = x$, where x is the top-left of the strip to be rendered.
3. Initialise $n = 0$, $z_0 = 0$.
   (a) Execute equation 4.1.
   (b) Increment n.
   (c) If $\mid z_n \mid^2 \geq 4$ then that c is not in the set of points which comprise the Mandelbrot set. Go to 4.
   (d) If $n > m$ then that c is in the Mandelbrot set, i.e. $c \subset M$. Go to 4.
   (e) Go to 3a.
4. Increment the real part of c. If the real part of c is less than the width of the strip to be rendered, go to 3.
5. Calculate the average of t and the time it took to render that line.
6. Set the real part of c to the left-hand of the strip. Increment the complex part of c. If the complex part of c is less than the height of the strip, go to 3.
7. Signal work completed, set $t = 0$ (thus this thread is guaranteed not to be selected by the work-stealing algorithm 5).
8. Suspend.

Algorithm 3 The work-stealing algorithm.

1. Monitor render threads for a work-completed signal. That thread that completes we shall denote as T_c.
2. Find that render thread with the longest estimated completion-time, t, note that each render thread updates this time upon completion of a line. Call this thread T_l.
3. Stop T_l when it completes the current line it is rendering.
4. Split the remaining work to be done in the strip equally between the two render threads T_c and T_l.
5. Restart the render threads T_c and T_l.
6. Go to 1.

very likely to have, the algorithm merely performs excessive work-stealing operations, but automatically tunes to find a local minimum in the total completion time curve. Experiments with [71] have shown that the algorithm can accommodate errors of over 100% in the estimated completion-times, and rapidly corrects to the new local minimum.

4.3 A Discussion of the Work-Stealing Algorithm 5

The algorithm 5 has some important features:

- The bandwidth of the single thread that implements that algorithm is the limiting factor in its ability to scale. Conversely, this algorithm is able to tolerate failures in render threads and is therefore robust. If a render thread stops responding, eventually it will be the slowest, unfinished render thread, and its work will be stolen. It is possible to scale this work-stealing algorithm, if one observes that the work-stealing algorithm operates upon a slice of the complex plane, demonstrating that the work-stealing algorithm is recursive. It is possible to assign strips $s_{0...j}$ of the plane to independent sets of render threads, governed by their own work-stealing thread. These s_i strips are monitored by a work-stealing thread in turn, those strips returning an aggregate estimated completion time. But this has a limitation: Once the number of render threads becomes of the order of the vertical resolution of the image, the completion time is bounded by the maximum time it takes a render thread to generate a single line. This line for the Mandelbrot set in figures 4.1 and 4.2 is the line $(-2, 0)$ to $(2, 0)$, which has the most points within the set. These points take m time to classify. As the unit of work in the work-stealing algorithm is a line, this is the slowest line, and thus the ultimate limit of this algorithm, unless the resolution is increased. This discussion leads to the following algorithm:

- If robustness is not required, then the image generated may be viewed as an array of values, where each of these values is the classification of c. Consider if there are $p_{0...q}$ threads, each p_n thread initially classifies a point in the array

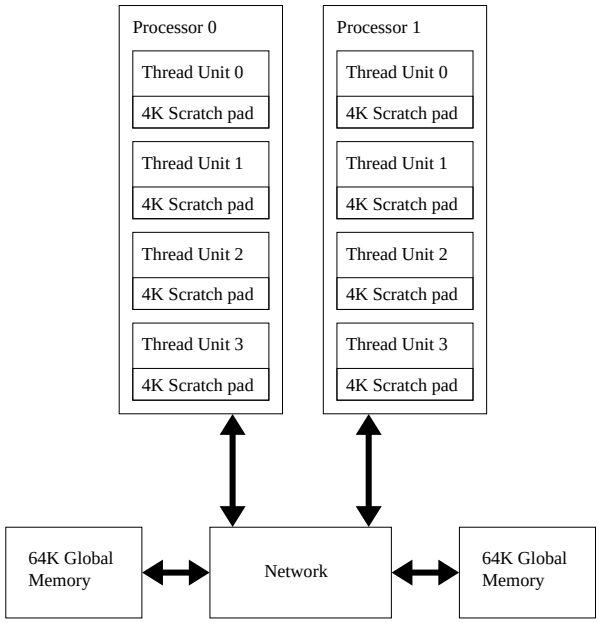


Figure 4.3: Simplified schematic overview of the DIMES/P implementation of CyclopsE.

offset by n, and once completed, moves along the array using a stride of q. This allows the use of a number of threads that is bounded by the number of points within the image. As this may be for an image of resolution 100×100, thus 10,000 points, which maps well on to the cellular architectures as described in [21]. For more thread units, the image resolution would need to be increased. Unfortunately, this algorithm does not have a natural ability to tolerate failures in thread units, unlike the work-stealing algorithm, 5.

4.4 DIMES/P Implementation of the Mandelbrot-set application

A simplified schematic diagram of the DIMES/P implementation (from [90]) of the CyclopsE processor is given in figure 4.3. The features of this architecture are that the memory model for the two types of memory, the scratch-pad and the global memories are different:

- Global memory obeys the Sequential Consistency Model for all thread units.

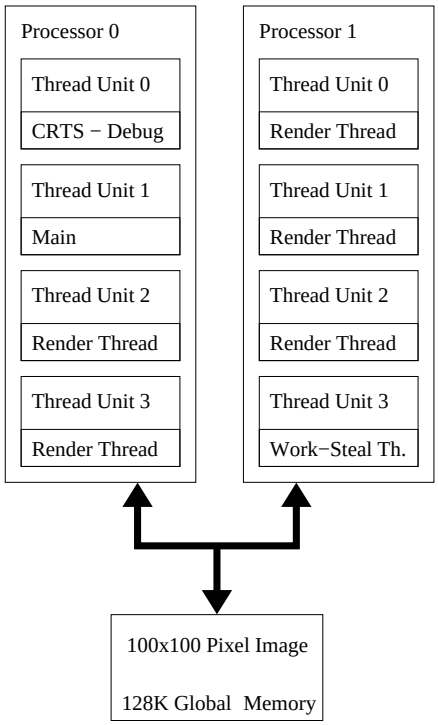


Figure 4.4: Layout of the render and work-stealing threads within the DIMES/P system.

- Scratch-pad memory obeys the program consistency model for all thread units, apart from the thread unit to which it is attached.

Such different consistency models affect the way that the data for the Mandelbrot-set application is arranged in memory, but this will be discussed in more detail in section 4.4.1.

The static layout of the render and work-stealing threads within the DIMES/P system is shown in figure 4.4. The software threads that occupy the thread units are:

- The *CRTS - Debug* thread is required for the debugger, if it is executed. As threads are statically allocated at program start-up, this must be left free for the debugger and Cellular Run-Time System (CRTS[2]) support.
- *Main* is the main loop of the Mandelbrot-set application.
- The *Render Threads* are the threads that execute algorithm 4.

[2]Not to be confused with the ANSI/ISO 'C' Runtime.

- *Work-Steal Th* is the thread that executes algorithm 5. Only one work-stealing thread was implemented, due to the limited number thread units per processor. In principle a render thread could also run on this thread unit, but the CRTS does not support virtual threads, moreover the work-stealing thread actually has to spin in a busy wait monitoring for completion of a render thread. Hence, for this application, it was deemed an unnecessary complexity.

Further details regarding the implementation may be found in [70].

4.4.1 The Memory Layout

As far as the programmer is concerned, the two 64k global memory units comprise a single, contiguous 128k block of memory which is for code and global data. The programmer has no access to differentiate between them. Moreover, the CyclopsE design is such that access times to them are the same, no matter which thread unit from which processor accesses them. The programmer may ensure that data will be placed in global memory by the compiler by ensuring that it is static. This may be done by making it global, or using the C/C++ keyword "static". The compiler places the stack frame into the scratch-pad memory, which means that function call depth is limited, as there is only 4K stack space per thread. The Mandelbrot-set algorithm as described does not need this much space for each thread unit, thus all thread local-data is placed into the corresponding scratch-pad memory for performance.

The 40,000 bytes of image data (100×100 words, 1 word = 4 bytes) is placed in global memory for implementation reasons. DIMES/P has no console, thus the only way that communication with DIMES/P can occur is via the global memory from a specially written program running on the host computer.

4.4.2 The Host Interface

The DIMES/P implementation is physically located on an FPGA on a PCI board, with specialized hardware and software support for it to communicate with the host computer for loading programs, and communicating results, of which details are given in [29, 90]. A simple command-line program was written to periodically

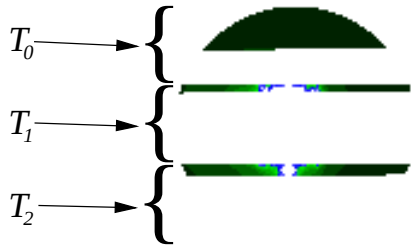


Figure 4.5: The image generated shortly after program start-up.

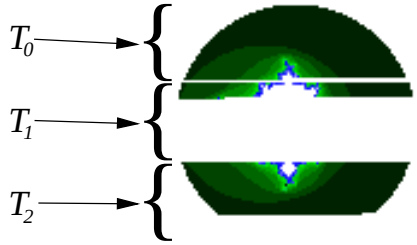


Figure 4.6: Image generation has progressed, shortly before a work-stealing event.

fetch the image data from the DIMES/P memory, and save it to a file for subsequent display. This program also allows the user to enter image parameter data for subsequent control of the image rendering on DIMES/P.

4.4.3 Execution details of the Mandelbrot-set application

In this example, there are three threads for simplicity:

1. On program start, the render threads start to execute and perform their assigned work. The assigned work is initially equal $\frac{1}{3}$ portions of the total image, arranged in horizontal strips. The top render thread is denoted by T_0, the middle by T_1 and the lower by T_2, although this relative position will change later. The operation of the render threads may be seen in figure 4.5. No work-stealing has occurred, so there are just three strips, one per render thread, scanning from left to right, top to bottom.

2. As the image generation proceeds, the T_0 and T_2 threads progress faster than T_1, as seen in figure 4.6. Note how T_0 has calculated more than T_2 - the lighter areas take longer to calculate, and the strip generated by T_0 is black at the top, and white at the bottom, but the converse is true for T_2.

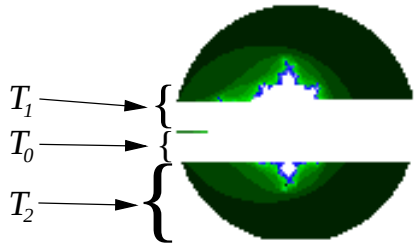


Figure 4.7: Just after the first work-stealing operation.

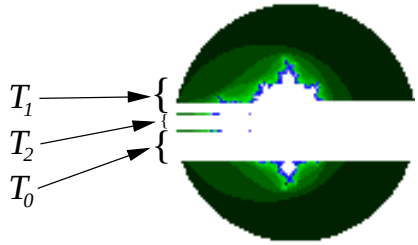


Figure 4.8: The second work-stealing operation.

3. The first work-stealing operation has just occurred. T_0 finished and T_1, the slowest (mainly white) has had the remaining work divided between it and T_0, see figure 4.7. Note how the end-point of T_1 was assigned to be the new end point of T_0 and the new end-point of T_1 is the start-point of T_0.

4. Shortly after this first work-stealing operation occurs, T_2 completes its assigned work. A second work-stealing operation occurs, see figure 4.8. In this case work was again stolen from T_1, and assigned to T_2.

5. After a pause T_1 completes its assigned work, and another work-stealing operation occurs, this time with T_0, which may be seen in figure 4.9.

6. Finally the set is completed, see figure 4.10 , with no further work-stealing operations, as the number of uncompleted lines for any render thread is less

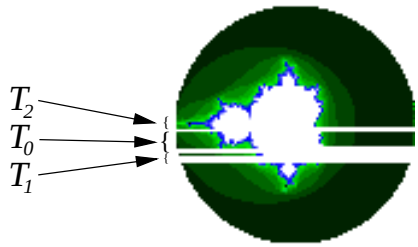


Figure 4.9: The third work-stealing operation.

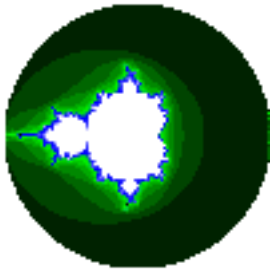


Figure 4.10: The completed Mandelbrot set.

than 2, and a line is the minimum unit of work for this algorithm.

Chapter 5

List of Achievements

The following goals have been achieved by the author in the course of the MSc(Res) program:

- A collaboration between the University of Hertfordshire and the CAPSL group, under Professor Gao at the University of Delaware, was set up by the author. As part of this collaboration the author worked at the CAPSL group for approximately 18 months.
- Two departmental seminars regarding this work were presented at CAPSL.
- A poster of the Mandelbrot set implementation on the DIMES/P-2 platform was shown at Super Computing '03, amongst other posters from the CAPSL group regarding DIMES.
- Two departmental seminars on regarding this work were presented at the University of Hertfordshire, shortly after the author's return from the CAPSL.
- A conference paper by the author has been accepted for the 11^{th} Asia-Pacific Computer Systems Architecture Conference, titled: "The Challenges of Efficient Code-Generation for Massively Parallel Architectures", also to be presented by the author. It is included in appendix C.

Chapter 6

Summary

The limitations of DIMES/P prevented further study of the properties of this program: scalability and timings were not done because of the limited number of thread units (8) and memory capacity (128k). Despite this, the development of the program was instructive: an initial contention of this thesis was that the memory models and massive parallelism (i.e. large numbers of micro-threads) inherent in cellular architectures would make programming for them hard. This was experienced to different measures, relating to the memory model support, the thread library and therefore the micro-thread support.

With regards to the memory model support, the fact that the compiler made natural use of language-level syntax to map data into scratch-pad and on-chip memory (using the C/C++ keyword *static*) made using these different memory models easy. But this simplicity was at a price: Cyclops only has word-sized, atomic memory-operations, and these operations were apparently unused for this problem. The author contends that such multiple, read-modify-write operations that must be maintained as an atomic unit hampered the performance of the program on Cyclops, because they could not make use of the hardware-level support for atomic operations. So the more usual barriers such as mutexes and critical sections were needed. This implies that the manual locking that had to be applied should really have been implemented within the compiler-provided support via the static keyword. If this were the case then it may have been possible for the compiler to perform optimization on the locking of access to the data, and improved program performance, with

apparently no impact upon the developer. As already mentioned, certain members of the C++ standards committee are proposing the extension of the execution model within the C++ standard to support the concepts of memory consistency within the C++ standard. That this proposal will address the problem outlined above is not yet clear. It is the author's contention that there should be support for such locking (in some manner implemented within the compiler or a run-time library) if programs more sophisticated than the one described in this thesis are to be successfully written for these architectures.

With regards to the thread library: the complexity of POSIX-Threads has been a hindrance to successful multi-thread program creation. Indeed this opinion has been voiced by some members of the C++ standards committee at the ACCU 2004, 2005 and 2006 conferences. The creation of a C++ wrapper to hide thread creation and destruction, and combine with that thread, any local storage in an efficient manner, was only partially successful: The concept that a thread is an object has not been not universally accepted, because this means that the data to be manipulated becomes intimately intermingled with the thread-management code. This would be an even greater problem when considering micro-threads in that they have little, or indeed no context, thus mingling threading constructs with data is potentially in direct conflict with the design of micro-threads. Even for this simple example program, this mingling was evident in the work-stealing algorithm, and the way it interacted with the start and end-points of the worker threads. For more complex, larger programs, such complexity would be likely to make writing them correctly, and modifying them later very hard. Subsequent updates to the cthread model, which became TNT, described in [31], are still largely POSIX-Thread based, and which is a low-level API. The concept that data and execution should be kept separate is commonly and naturally embodied in programming via the syntax of "main". It has been contended by members of the C++ Standards committee that this pattern should be duplicated for thread libraries: that there should exist a pool of threads, to which work is passed. This work would be asynchronously executed, on a thread within the pool. With the results returned from that pool via a wait-able object. This concept is similar to the data-flow designs that preceded VLIW, indeed it has

been argued that this concept is a software emulation of data-flow.

When considering the harder problem of creating an effective algorithm to implement micro-threading and representing that in code, clearly the example program described above was very limited in its achievements. The work-stealing algorithm was intimately related to the program design. The ability to abstract the work-stealing operation to other problems would be very limited using that design. Alternative approaches have been examined, such as in [88], using OpenMP, which was still used as a library to express the parallelism, but OpenMP poorly maps to micro-threads, the primitives it implements, arguably, have been too tied into the process-level parallelism for which it was originally designed. Alternatively, if one is to consider the suggestion above, of a micro-thread pool into which work is submitted, then the details of how the pool works become separated from the work itself. The fact that the pool balances work between threads using a master-slave, or work-stealing algorithm should be independent of the work: a natural division of concepts. If this were the case, then the programmer would be free to add work to the pool as desired. The parallelism of the algorithm would be more naturally expressed in terms of operations on data. If one is to consider this further: the actual executable code (in terms of the function pointer, in micro-threading terms a program counter) and the data are passed to the pool together. It could be then possible for the pool to be designed to make use of data locality and code locality: Did a previous thread execute that code before? If so, prefer to run that work on that thread. If there are "cliques" of threads, related due to resource asymmetry, then one might create a pool to represent the particular feature of that resource. For example a Cyclops chip might be represented as a micro-thread pool, contained within a greater thread pool that represents the machine, due to the fact that off-chip memory access makes use of a message-passing protocol, rather than the crossbar network embedded within the chip, that allows much more rapid memory access.

It is still an open question regarding what may be the ideal approach to implementing parallelism via micro-threading: language-level support such as UPC, HPF or other language extensions, or within the compiler using trace-scheduling, or should it be at a library-level using, for example OpenMP or POSIX-Threads, or

should it be within the architecture, such as the micro-threaded architectures [13] of Luo et al [68], CARE [75] or Cyclops [31].

Bibliography

[1] Adam, T.L., Chandy, K.M. and Dickson, J.R., "A Comparison of List Schedules for Parallel Processing Systems," CACM, 17, 12, pp. 685-690, December, 1974.

[2] Ahuja, R.K., Magnanti, T.L. and Orlin, J.B., "Network Flows: Theory, Algorithms and Applications," Prentice-Hall, 1993.

[3] Allen, J.R., Kennedy, K., Porterfield, C. and Warren, J.D., "Conversion of Control Dependence to Data Dependence," Proceedings of the 10^{th} ACM Symposium on Principles of Programming Languages, pp. 177-89, January 1983.

[4] Almásil, G., Cascaval, C., Castaños, J.G., Denneau, M., Lieber, D., Moreira, J.E. and Warren, H.S., "Dissecting Cyclops: Detailed Analysis of a Multithreaded Architecture.", ACM SIGARCH Computer Architecture News, Vol. 31, March 2003

[5] Amaral, J.N., Gao, G.R., and Tang, X., "An Implementation of a Hopfield Network Kernel on EARTH," CAPSL Technical Paper, 1998. `http://www.capsl.udel.edu`

[6] Aono, M., Kunil, T.L., "Botanical Tree Image Generation.", IEEE Computer Graphics and Applications 4,5 (1984) 10-33.

[7] Architecture Simulation Framework, `http://www.lri.fr/~osmose/ASF/`, latest updated in 28/06/2001 or `http://www-rocq.inria.fr/a3/tools.html.en`.

[8] Avarind, R.S.N., and Pingail, K.K., "I-structures: Data Structures for Parallel Computing," ACM TOPLAS, 11(4): pp. 598-632, October 1989.

[9] Backus, J., "Can programming be liberated from the von Neumann style? A functional style and its algebra of programs," Communications of the ACM 21, 8, pp. 613-641, August 1978.

[10] Barnsley, M.F., Devaney, R.L., Mandelbrot, B.B., Peitgen, H.-O., Saupe, D., Voss, R.F., "The Science of Fractal Images.", Springer-Verlag, 1988.

[11] G.E. Blelloch, P.B. Gibbons, Y. Matias and G.J. Narlikar, "Space-Efficient Scheduling of Parallelism with Synchronization Variables," Proceedings of the 9th ACM Symposium on Parallel Algorithms and Architectures (SPAA), June 1997.

[12] Bohm A.P.W. and Sargeant, J., "Efficient Dataflow Code Generation for SISAL", IEEE Transactions on Computers, vol.C-38 no.1, pp. 4-14, January 1989.

[13] Bolychevsky, A., Jesshope, C.R. and Muchnick, V.B., "Dynamic Scheduling in RISC Architectures," IEE. Proc.-Comput. Digit. Tech., Vol. 143, No. 5, Sept. 1996.

[14] Bruening, U., Giloi, W.K. and Schroeder-Preikschat, W., "Latency Hiding in Message Passing Architectures", Proceedings of the 8th International Parallel Processing Symposium [21], pp. 704-709.

[15] Burger, D., "Memory Bandwidth Limitations of Future Microprocessors.", ISCA 1996.

[16] Burks, A.W., Goldstine, H.H. and von Neumann, J., "Preliminary discussion of the logical design of an electronic computing instrument." A.H. In Taub, editor, John von Neumann Collected Works, The Macmillan Co., New York, Volume V, pp. 34-79, 1963.

[17] Burtscher, M. and Zorn, B.G., "Prediction Outcome History-based Confidence Estimation for Load Value Prediction," The Journal of Instruction-Level Parallelism, vol. 1, February 1999. `http://www.jilp.org/vol1`

[18] Cai, H., "Dynamic Load-Balancing on the EARTH-SP System.", Master's Thesis, McGill University, Montréal, May 1997.

[19] B. Calder and D. Grunwald, "Reducing Indirect Function Call Overhead in C++ Programs," Proceedings of the 21st Symposium on The Principles of Programming Languages, pp. 397-408, January, 1994.

[20] Calder, B., Feller, P. and Eustace, A., "Value Profiling and Optimization," The Journal of Instruction-Level Parallelism, vol. 1, February 1999. `http://www.jilp.org/vol1`

[21] Cascaval, C., Castaños, J.G., Ceze, L., Denneau, M., Gupta, M., Lieber, D., Moreira, J.E., Strauss, K. and Warren, H.S., "Evaluation of a Multithreaded Architecture for Cellular Computing.", 8th International Symposium on High-Performance Computer Architecture (HPCA), February 2002.

[22] Cavalherio, G.G.H., Doreille, M., Galilée, F., Gautier, T., Roch, J-L., "Scheduling Parallel Programs on Non-Uniform Memory Architectures.", HPCA Conference – Workshop on Parallel Computing for Irregular Applications WPCIA1, Orlando, USA, January 1999.

[23] Chang, P.-Y., Hao, E., Yeh, T.-Y. and Patt, Y.N., "Branch Classification: A New Mechanism for Improving Branch Predictor Performance," Proceedings of MICRO-27, pp. 22-31, Nov-Dec 1994.

[24] Chappell, R.S., Stark, J., Knott, S.P., Reinhardt, S.K. and Patt, Y.N., "Simultaneous Subordinate Microthreading *(sic)*," Proceedings of the 26th International Symposium on Computer Architecture, IEEE, 1998.

[25] Cleary, J. and Witten, I., "Data Compression using Adaptive Coding and Partial String Machines," IEEE Transactions on Communications, vol. 32, pp. 396-402, April 1984.

[26] Cmelik, B. and Keppel, D., "Shade: A Fast Instruction-Set Simulator for Execution Profiling," ACM SIGMETRICS Conference on Measurement and Modelling of Computer Systems, 1994.

[27] Coffman, J.R., ed., "Computer and Job-Shop Scheduling Theory," John Wiley, New York, 1976.

[28] Cohn, R. and Lowney, P.G., "Design and Analysis of Profile based Optimization in Compaq's *(sic)* Compilation Tools for the Alpha," The Journal of Instruction-Level Parallelism, vol. 2, January 2000. `http://www.jilp.org/vol2`

[29] del Cuvillo, J.B., Klosiewicz, R. and Zhang, Y., "A Software Development Kit for DIMES.", CAPSL Technical Note 10, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, May 2003, `ftp://ftp.capsl.udel.edu/pub/doc/notes/`.

[30] del Cuvillo, J.B., Zhu, W., Hu, Z. and Gao, G.R., "FAST: A Functionally Accurate Simulation Toolset for the Cyclops-64 Cellular Architecture.", Workshop on Modeling, Benchmarking and Simulation (MoBS), held in conjunction with the 32nd Annual International Symposium on Computer Architecture (ISCA'05), Madison, Wisconsin, June 4, 2005.

[31] del Cuvillo, J.B., Zhu, W., Hu, Z. and Gao, G.R., "TiNy Threads: a Thread Virtual Machine for the Cyclops64 Cellular Architecture.", Fifth Workshop on Massively Parallel Processing (WMPP), held in conjunction with the 19th International Parallel and Distributed Processing System, Denver, Colorado, April 3 - 8, 2005.

[32] Dachselt, F., Kelber, K. and Schwarz, W., "Chaotic Coding and Cryptanalysis.", Proceedings of 1997 IEEE International Symposium on Circuits and Systems Circuits and Systems in the Information Age (New York, USA), vol. 4, May 1998, pp. 518-21.

[33] Duller, A., Towner, D., Panesar, G., Gray, A. and Robbins, W., "picoArray technology: the tool's story.", Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, IEEE, 2005.

[34] Egan, C., Steven, G. and Vintan, L., "Cached Two-level Adaptive Branch Predictors with Multiple Stages," In Trends in Network and Pervasive Computing

ARCS 2002 (Lecture Notes in Computer Science 2299), Springer-Verlag, pp. 179 191, 2002.

[35] Eggers, S.J., Emer, J.S., Levy, H.M., Lo, J.L., Stamm, R.L. and Tullsen, D.M., "Simultaneous Multithreading *(sic)*: A Platform for Next-Generation Processors," IEEE Micro, Vol. 17, No. 5, September/October 1997.

[36] Emami, M, Ghiya, R. and Hendren, L.J., "Context-sensitive Interprocedural *(sic)* Points-to Analysis in the Presence of Function Pointers," Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation, SIGPLAN Notices 29(6), pp. 242-56, June 1994.

[37] Engler, D.R., Andrews, G.R. and Lowenthal, D.K., "Filaments: Efficient Support for Fine-Grain Parallelism.",TR 93-13a, Dept. of Computer Science, University of Arizona, Tucson, 1993.

[38] Fisher, J.A., "The Optimization of Horizontal Microcode within and beyond Basic Blocks: An Application of Processor Scheduling with Resources,", PhD dissertation, University of New York, New York, 1979.

[39] Gao, G.R., Theobald, K.B., Márquez, A., and Sterling, T., "The HTMT Program Execution Model," CAPSL Technical Memo 9, July 1997. `.http://www.capsl.udel.edu`

[40] Gao, G.R. and Sarkar, V., "Location Consistency - a New Memory Model and Cache Consistency Protocol.", IEEE Transactions on Computers, Vol. 49, No. 8, August 2000.

[41] Gao, G.R., Theobald, K.B., Hu, Z., Wu, H, Lu, J., Sterling, T.L., Pingali, K., Stodghill, P., Stevens, R. and Hereld, M., "Next Generation System Software for Future High-End Computing Systems.", International Parallel and Distributed Processing Symposium: IPDPS 2002 Workshops April 15 - 19, 2002 Fort Lauderdale, Florida.

[42] Gao, G.R., Theobald, K.B., Govindarajan, R., Leung, C., Hu, Z., Wu, H., Lu, J., del Cuvillo, J., Jacquet, A., Janot, V. and Sterling, T.L., "Program-

ming Models and System Software for Future High-End Computing Systems: Work-in-Progress.", International Parallel and Distributed Processing Symposium (IPDPS'03) April 22 - 26, 2003 Nice, France.

[43] Garey, M.R. and Johnson, D.S., "Computers and Intractability: A Guide to the Theory of NP-Completeness," W.H. Freemann and Co., New York, 1979.

[44] Garfinkel, A., Chen, P.S., Walter, D.O., Karagueuzian, H.S., Kogan, B., Evans, S.J., Karpoukhin, M., Hwang, C., Uchida, T., Gotoh, M., Nwasokwa, O., Sager, P. and Weiss, J.N., "Quasiperiodicity and chaos in cardiac fibrillation.", Journal of Clinical Investigation. 99(2), pp. 305-14, January 1997.

[45] El-Ghazawi, T.A., Carlson, W.W., Draper, J.M., "UPC Language Specifications V1.1.1", October 2003.

[46] Ghiya, R. and Hendren, L.J., "Connection Analysis: A Practical Interprocedural *(sic)* Heap Analysis for C," International Journal of Parallel Programming, 24(6), December 1996.

[47] Gottleib, A., Lubachevsky, B.D., and Rudolph, L., "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperating Sequential Processors," ACM Transactions on Programming Languages and Systems, 5(2), April 1983.

[48] Gurd, J.R. and Snelling, D.F., "Manchester Data-Flow: A Progress Report", ACM Proceedings of the 6th International Conference on Supercomputing, pp. 216-225, 1992.

[49] Hennessy, J.L. and Patterson, D.A., " Computer Architecture: A Quantitative Approach," 2^{nd} Edition, Morgan Kaufmann, 1996.

[50] Hoogerbrugge, J. and Augusteijn, L., "Instruction Scheduling for TriMedia," The Journal of Instruction-Level Parallelism, vol. 1, February 1999. `http://www.jilp.org/vol1`

[51] Hsu, P.Y.T. and Davidson, E.S., "Highly Concurrent Scalar Processing," Proceedings of the 13th Annual International Symposium on Computer Architecture, pp. 386-95, June 1986.

[52] Huang, J., and Lilja, D.J., "An Efficient Strategy for Developing a Simulator for a Novel Concurrent Multi-Threaded Processor Architecture," 1998.

[53] Hum, H.H.J., Maquelin, O., Theobald, K.B., Tian, X., Gao, G.R. and Hendren, L.J., "A Study of the EARTH-MANNA Multithreaded System.", Intl. J. of Parallel Programming, 24(4):319-347, August 1996.

[54] Hwu, W.W., Conte, T.M. and Chang, P.P., "Comparing Software and Hardware Schemes for Reducing the Cost of Branches," Proceedings of the 16th Annual International Symposium on Computer Architecture, pp. 224-233, May 1989.

[55] Hwu, W.W., Mahlke, S.A., Chen, W.Y., Chang, P.P., Warter, N.J., Bringmann, R.A., Ouellette, R.G., Hank, R.E., Kiyohara, T., Haab, G.E., Holm J.G. and Lavery, D.M., "The Superblock *(sic)*: An Effective Technique for VLIW and Superscalar Compilation," The Journal of Supercomputing, Vol. 7, 1/2: pp. 229-248, 1993.

[56] Intel Pentium 4 Processor Product Brief, http://www.intel.com/design/Pentium4/prodbref/ `http://www.intel.com/design/Pentium4/prodbref/`.

[57] Jagannathan, R., "Dataflow *(sic)* Models," E.Y. Zomaya, editor, Parallel and Distributed Computing Handbook, McGraw-Hill, 1985.

[58] Jesshope, C.R. and Luo, B., "Evaluation of Vector-Instruction Set Micro-Threaded Pipelines," Institute of Information Sciences and Technology, Massey University, New Zealand, private communications.

[59] Johnson, M., "Superscalar Microprocessor Design.", Prentice Hall, New Jersey, 1991.

[60] Julia, G., "Sur l'iteration des Functions Rationelles.", Journal de Math. Pure et Appl. 8 (1918) 47-245.

[61] Kakulavarapu, K.P., "Dynamic Load-Balancing Issues in the EARTH Runtime System.", Master's Thesis, McGill University, Montréal, April 2000.

[62] Kakulavarapu, P., Morrone, C.J., Theobald, K., Amaral J.N. and Gao, G.R., "A Comparative Performance Study of Fine-Grain Multi-threading on Distributed Memory Machines.", 19th IEEE International Performance, Computing and Communication Conference-IPCCC2000, Phoenix, Arizona, USA, Feb. 20-22, 2000.

[63] Kalamatiano, J. and Kaeli, D.R., "Indirect Branch Prediction using Data Compression Techniques," The Journal of Instruction-Level Parallelism, vol. 2, December 1999. `http://www.jilp.org/vol1`

[64] Kogge, P.M., Sterling, T.L. and Gao, G., "Processing in memory: Chips to petaflops.", In Workshop on Mixing Logic and DRAM: Chips that Compute and Remember at ISCA '97. `http://iram.cs.berkeley.edu/isca97-workshop/`, Denver, CO, June 1997.

[65] Lam, M.S. and Wilson, R.P., "Limits of Control Flow on Parallelism," Proceeding of the 19th Annual International Symposium on Computer Architecture, ACM, May 1992, pp. 46-57.

[66] Lee, C., Potkonjak, M. and Mangione-Smith, W.H., "Mediabench *(sic)*: A Tool for Evaluating and Synthesizing Multimedia and Communications Systems," Proceedings of the 30th Annual International Symposium on Micro-architecture, December 1998.

[67] Lo, J. L., Eggers, S. J., Emer, S. J., Levy, H. M., Stamm, R. L., Tullsen, D. M., "Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading," ACM Transactions on Computer Systems, Vol. 15, No. 3, August 1997, Pages 322354.

[68] Luo, B., and Jesshope, C.R., "Performance of a Micro-Threaded Pipeline," ACSAC 2002, Melbourne, Australian, Vol. 6.

[69] McFarling, S., "Combining Branch Predictors," Tech. Note TN-36, DEC WRL, June 1993.

[70] M^{c}Guiness, J.M., "A DIMES Demonstration Application: Mandelbrot-Set Generation Using a Work-Stealing Algorithm.", CAPSL Technical Note 11, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, June 2003, `ftp://ftp.capsl.udel.edu/pub/doc/notes/`.

[71] M^{c}Guiness, J.M., "Aleph One", `http://aleph1.sourceforge.net/`.

[72] Mandelbrot, B.B., "The Fractal Geometry of Nature.", W.H.Freeman & Co., Sept., 1982.

[73] Márquez, A, Theobald, K.B., Tang, X. and Gao, G.R., "A Superstrand *(sic)* Architecture," CAPSL Technical Memo 14, December 1997. `http://www.capsl.udel.edu`

[74] Márquez, A, Theobald, K.B., Tang, X., Sterling, T. and Gao, G.R., "A Superstrand *(sic)* Architecture and its Compilation," CAPSL Technical Memo 18, March 1998. `http://www.capsl.udel.edu`

[75] Márquez, A., "CARE Architecture," PhD dissertation, University of Delaware, 2004.`http://www.capsl.udel.edu/publications.shtml##5`

[76] Moreira, J.E., "On the Implementation and Effectiveness of Autoscheduling for Shared-Memory Multi-Processors," PhD Thesis, Univerisity of Illinois, Urbana, Illinois, USA, 1995.

[77] Morrone, C.J., Amaral, J.N., Tremblay, G. and Gao, G.R., "A Multi-Threaded Runtime System for a Multi-Processor/Multi-Node Cluster.", 15th Annual International Symposium on High Performance Computing Systems and Applications, June 18-20, 2001, Windsor, ON, Canada.

[78] Moshovos, A. and Sohi, G.S., "Memory Dependence Prediction in Multimedia Applications," The Journal of Instruction-Level Parallelism, vol. 2, January 2000. `http://www.jilp.org/vol2`

[79] Muchnick, S.S., "Advanced Compiler Design and Implementation," Morgan Kaufmann Publishers, San Francisco, 1997.

[80] Patterson, D.A. and Hennessy, L.J., "Computer Architecture: A Quantitative Approach.", 2nd Edition, Morgan Kaufmann Inc., San Francisco, pp. 374, 1996.

[81] "Perfect Developer", `http://www.eschertech.com/index.html`

[82] Peters, E.E., "Fractal Market Analysis : Applying Chaos Theory to Investment and Economics.", John Wiley & Sons, 1994.

[83] Postiff, M.A., Greene, D.A., Tyson, G.S. and Mudge, T.N., "The limits of instruction level parallelism in SPEC95 applications.", The Third Workshop on the Interaction between Compilers and Computer Architectures (INTERACT), in conjunction with the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VIII), October 1998.

[84] Postiff, M., Tyson, G. and Mudge, T., "Performance Limits of Trace Caches," The Journal of Instruction-Level Parallelism, vol. 1, February 1999. `http://www.jilp.org/vol1`

[85] Reghbati, H.K., "An Overview of Data Compression Techniques.", Computer 14,4 (1981) 71-76.

[86] Reilly, J., "SPEC Describes SPEC95 Products And Benchmarks", Intel Corporation, September 1995. `http://open.specbench.org/osg/cpu95/news/cpu95descr.html`

[87] Norton Riley, H., "The von Neumann Architecture of Computer Systems," Computer Science Department California State Polytechnic University Pomona, California, September 1987. `http://www.csupomona.edu/~hnriley/www/VonN.html`

[88] Rodenas, D., Martorell, X., Ayguade, E., Labarta, J., Almasi, G., Cascaval, C., Castanos, J. and Moreira, J., "Optimizing NANOS OpenMP for the IBM Cyclops

Multithreaded Architecture.", 19th IEEE International Parallel and Distributed Processing Symposium, Vol. 1, pp. 110, 2005.

[89] Rotenburg, E. and Smith, J.E., "Control Independence in Trace Processors," The Journal of Instruction-Level Parallelism, vol. 2, January 2000. `http://www.jilp.org/vol2`

[90] Sakane, H., Yakay, L., Karna, V., "DIMES/P Hardware Technical Manual.", CAPSL Technical Note 12, Department of Electrical and Computer Engineering, University of Delaware, Newark, Delaware, June 2003, `ftp://ftp.capsl.udel.edu/pub/doc/notes/`.

[91] Sakane, H., Yakay, L., Karna, V., Leung, C. and Gao, G.R., "DIMES: An Iterative Emulation Platform for Multiprocessor-System-on-Chip Designs.", IEEE International Conference on Field-Programmable Technology, December 15-17, 2003, Tokyo, Japan.

[92] Sankaralingam, K., Nagarajan, R., Liu, H., Kim, C., Huh, J., Burger, D., Keckler, S. W., Moore, C.R., "Exploiting ILP, TLP, and DLP with the Polymorphous TRIPS Architecture", 30$^{\text{th}}$ Annual International Symposium on Computer Architecture, 2003.

[93] Saulsbury, A., Pong, F., and Nowatzyk, A., "Missing the Memory Wall: The Case for Processor/Memory Integration.", In Proceedings of the 23$^{\text{rd}}$ International Symposium on Computer Architecture, pp. 90-101, May 1996.

[94] Savari, S. and Young, C., "Comparing and Combining Profiles," The Journal of Instruction-Level Parallelism, vol. 2, January 2000. `http://www.jilp.org/vol2`

[95] CAPSL Exhibit Booth 130, November 18$^{\text{th}}$-20$^{\text{th}}$ `http://www.capsl.udel.edu/dimes/sc2003_flyer.html`, as part of Super Computing 2003, [96].

[96] Super Computing 2003, November 15$^{\text{th}}$-21$^{\text{st}}$, Phoenix, Arizona, U.S.A.`http://www.sc-conference.org/sc2003/`

[97] Schnarr, E.C., "Applying Programming Language Implementation Techniques to Processor Simulation," PhD dissertation, University of Winsconsin, Madison, 2000.

[98] Sechrest, S., Lee, C.C. and Mudge, T., "The Role of Adaptivity in Two-level Adaptive Branch Prediction," 28th International Symposium on Microarchitecture, 1995.

[99] Sharp, J.A., "Data Flow Computing," Ellis Horwood Limited, Chichester, England, 1985.

[100] Skadron, K., Martonorst, M. and Clark, D.W., "Speculative Updates of Local and Global Branch History," The Journal of Instruction-Level Parallelism, vol. 2, December 1999. `http://www.jilp.org/vol2`

[101] Smith, J.E., and Sohi, G.S., "The Microarchitectures of Superscalar Processors.", In the Proceedings of the IEEE 1995.

[102] de Souza, A.F., and Rounce, P., "On the Effectiveness of the Scheduling Algorithm of the Dynamically Trace Scheduled VLIW Architecture," 11th Symposium on Computer Architecture and High Performance Computing, SBAC-PAD, 1999.

[103] Srinivasan, S.T. and Lebeck, A.R., "Load Latency Tolerance in Dynamically Scheduled Processors," The Journal of Instruction-Level Parallelism, vol. 1, February 1999. `http://www.jilp.org/vol1`

[104] Sterling, T., Becker, D.J., Savarese, D., Berry, M., and Res, C., "Achieving a Balanced Low-Cost Architecture for Mass Storage Management through Multiple Fast Ethernet Channels on the Beowulf Parallel Workstation", Proceedings of the International Parallel Processing Symposium, 1996. `http://cesdis.gsfc.nasa.gov/beowulf/papers/papers.html`

[105] Sterling, T.L., "An Introduction to the Gilgamesh PIM Architecture." Proc. European Conference on Parallel Processing, Manchester, UK, pp. 16-32, August 2001.

[106] Sterling, T.L., Zima, H.P., "Gilgamesh: A Multithreaded Processor-In-Memory Architecture for Petaflops Computing.", Proc.SC2002, Baltimore, November 2002.

[107] Steven, G., "Exploiting Instruction-Level Parallelism in High Performance Processors," Department of Computer Science, University of Hertfordshire, Unpublished, 2001.

[108] Stoutchinin, A., Amaral, J.N, Gao, G.R., Dehnert, J. and Jain, S., "Automatic Pre-fetching of Induction Pointers for Software Pipelining," CAPSL Technical Memo 37, November 1999. `http://www.capsl.udel.edu`

[109] Tang, X., "Compiling for Multithreaded *(sic)* Architectures," PhD dissertation, University of Delaware, Autumn 1999.

[110] Tarlescu, M.D., Theobald, K.B. and Gao, G.R., "Elastic History Buffer: A Low-Cost Method to Improve Branch Prediction Accuracy," IEEE Conference on Computer Design, October 1997.

[111] Tate, D., "Superscalar Architectures and Statically Scheduled Programs," PhD dissertation, University of Hertfordshire, Hatfield, Hertfordshire, U.K., 2000.

[112] Theobald, K. B., Gao, G. R. and Hendren, L. J, "Speculative Execution and Branch Prediction on Parallel Machines," ICS-7/93, ACM, 1993.

[113] Tullsen, D.M., Eggers, S.J. and Levy, H.M., "Simultaneous Multithreading *(sic)*: Maximising *(sic)* On-chip Parallelism," Proceedings of the 22nd Annual International Symposium on Computer Architecture, pp 392-402, June 1995.

[114] Turcotte, D.L., "Fractals and Chaos in Geology and Geophysics.", Cambridge University Press, 1992, pp. 35-50.

[115] Unger, A., Ungerer, Th. and Zehender, E., "Simultaneous Speculation Scheduling," 11th Symposium on Computer Architecture and High Performance Computing, SBAC-PAD '99, 1999.

[116] Waingold, E., Taylor, M., Srikrishna, D., Sarkar, D., Lee, W., Lee, V., Kim, J., Frank, M., Finch, P., Barua, R., Babb, J., Amarasinghe, S. and Agarwal, A., "Baring It All to Software: RAW Machines.", Computer September 1997 (Vol. 30, No. 9) pp 86 -93.

[117] Wall, D.W., "Limits of Instruction-Level Parallelism," Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating System, SIGPLAN Notices, vol. 26, no. 4, ACM Press, New York, NY, pp. 176-189, 1991.

[118] Wang, Z., Pierce, K. and McFarling, S., "BMAT – A Binary Matching Tool for Stale Profile Propagation," The Journal of Instruction-Level Parallelism, vol. 2, January 2000. `http://www.jilp.org/vol2`

[119] Wegner, T., Osuch, J., Martin, G., Bussell, B., "Fractint", `http://www.fractint.org/`

[120] Woo, S.C., Ohara, M., Torrie, E., Singh, J.P. and Gupta, A., "The SPLASH-2 programs: characterization and methodological considerations.", In Proceedings of the 22nd Annual International Symposium on Computer Architecture, ACM/IEEE, Portofino, Italy, pp. 24-36, 1995.

[121] Wulf, W. and McKee, S., "Hitting the memory wall: Implications of the obvious.", Computer Architecture News, 23(1), pp. 20-24, 1995.

[122] Yeh, T.-Y., and Patt, Y.N., "Alternative Implementations of Two-Level Adaptive Branch Prediction," Proceedings of the 19th Annual International Symposium on Computer Architecture, pp. 124-34, May 1992.

[123] Yeh, T.-Y., and Patt, Y.N., "A Comparison of Dynamic Branch Predictors that use Two Levels of Branch History," Proceedings of the 20th Annual International Symposium on Computer Architecture, pp. 257-66, May 1993.

[124] Zhang, Y., Zhu. W., Chen, F., Hu, Z. and Gao, G.R, "Sequential Consistency Revisited: The Sufficient Conditions and Method to Reason Consistency Model of a Multiprocessor-on-a chip Architecture.", The IASTED International

Conference on Parallel and Distributed Computing and Networks (PDCN2005), February 15 - 17, 2005, Innsbruck, Austria.

Appendix A

Implementing Applications on a Cellular Architecture - the Mandelbrot-set.

This was a draft paper prepared at CAPSL, at the University of Delaware in collaboration with Dr. Egan, for submission to various conferences, before the author left Delaware.

Authors: Jason M^{c}Guiness[1,2], Colin Egan[2], Guang Gao[1].

[1]University of Delaware, Newark, DE.

[2]University of Hertfordshire, Hatfield, Hertfordshire, U.K. AL10 9AB.

mcguines@capsl.udel.edu

c.egan@herts.ac.uk

ggao@ee.udel.edu

A.1 Abstract.

There is an ever widening gap between CPU speed and memory speed, resulting in a 'memory wall' where the time for memory accesses dominate performance. Cellular architectures, such as the Cyclops family, have been developed to overcome this 'memory wall' by implementing processors-in-memory (PIM) on the same chip. PIM

architectures achieve high performance by increasing the bandwidth of processor-memory communication and reducing latency. In this paper we introduce DIMES (the Delaware Iterative Multiprocessor Emulation System) which is being developed by CAPSL at the University of Delaware, as a hardware validation tool for cellular architectures. The version of DIMES used in this paper is a simplified hardware implementation of the Cyclops-64 cellular architecture developed at the IBM T. J. Watson Research Center. Since DIMES is a hardware validation tool, its hardware implementation is constrained to a dual processor where each processor has four thread units. DIMES memory is restricted to 16K of local scratch-pad memory per processor and 64K global shared memory. Additionally DIMES is linked to a host computer for I/O. We have chosen to use a Mandelbrot-set generator (written in C++) with a work-stealing algorithm as our metric to evaluate the programming model on DIMES. The Mandelbrot-set generator has been threaded, and the work-stealing algorithm achieves load balancing between the DIMES' threads. The Mandelbrot example demonstrates the effective use of DIMES' threads, the effective use of DIMES scratch-pad memory and the effective use DIMES global memory in its CRTS environment. The results of the study are highly promising and show that DIMES is an ideal hardware tool for validating future Cyclops enhancements.

A.2 Introduction.

High performance processors, in particular super-scalars, exploit instruction level parallelism (ILP) by overlapping instruction execution (pipelining) and using multiple instruction issue (MII) per clock cycle [101]. Although, this approach improves processor performance, it does not improve performance of the memory subsystem. Researchers improve CPU speed by increasing the number of instructions issued in each clock cycle or by increasing the depth of the pipeline, which can cause a bottleneck in the memory-subsystem. This is termed as the memory-wall and impacts on overall system performance [121].

One approach to overcome the memory-wall is to improve data throughput and data storage between the memory subsystem and the CPU by introducing extra

levels in the memory hierarchy [15, 121]. However, introducing extra levels in the memory hierarchy increases the penalty associated with a miss in the memory-subsystem, which limits the amount of ILP and impacts on processor performance. Also, there is an increase in design complexity and an increase in power consumption of the overall system. Furthermore, increasing the number of levels in the memory hierarchy does not improve memory access times.

An alternative approach to overcome the memory-wall is to improve both data-processing and data-access time by the integration of processing logic in memory [21, 41, 105, 106, 116]. The idea of integrating processors-in-memory (PIMs) is to simplify the memory hierarchy design, to achieve higher bandwidth and to reduce latency. There are several PIM architectures being developed, for example, the Cyclops family of PIM architectures by IBM [21], the Gilgamesh PIM architecture by NASA [105, 106], the polymorphous TRIPS architecture at Austin, Texas [92] and the Shamrock PIM architecture at Notre Dame, France [64].

A problem with integrating a processor and memory on in the same silicon space is that the processor speed is reduced in comparison with a high performance processor and the amount of memory is also reduced [21]. To overcome the reduction in processing power and the reduction in the amount of available memory and therefore latency, multiple PIM chips are connected together forming a network of cells, where a single PIM chip is considered to be a cell and the whole architecture is described as cellular.

To overcome the data access problem, each cell is threaded such that each thread unit is independent from all other thread units. In this multi-threaded organisation, every thread unit serves as an independent single-issue in-order processor, which shares computationally expensive hardware resources such as floating-point units and caches.

In this paper we introduce DIMES (the Delaware Iterative Multiprocessor Emulation System) which is being developed by CAPSL at the University of Delaware [29,90]. DIMES is a hardware validation tool for cellular architectures, in particular the Cyclops family [21]. DIMES places the Cyclops architectural design into a single FPGA. The idea behind DIMES is to emulate Cyclops cycle by cycle, to be far

faster than software based simulations, and to direct future Cyclops enhancements.

A.3 Programming Models on Cellular Architectures.

Cellular architectures require different programming models to the general-purpose code executed by super-scalar processors [40, 41, 120]. Gao proposes the use of a combination of execution models and memory models, because of the cellular architectures multiple execution units within each cell.

Gao's programming model evaluates multiple threads in each cell due to the large number of execution units within Cyclops. For example, one programming model uses thread percolation as a technique to perform dynamic load-balancing [18,53,62]. Additionally, in cellular architectures, multiple threads perform memory accesses independently. As a result of this, the memory subsystem requires some form of access model that allows these memory references to be effectively served. For example, the use of the location-consistency model was suggested as a memory access model by [40].

A.4 Conclusion/Discussion.

The threaded algorithm shows that the Mandelbrot set is an ideal mechanism for evaluating cellular architectures and programming models on the DIMES hardware. Currently DIMES is targeted towards CyclopsE, however DIMES could be expanded to the full IBM Cyclops family and other cellular architectures, such as those at Gilgamesh at NASA and Shamrock at Notre Dame.

Future enhancements to DIMES may incorporate more hardware to allow benchmarks, such as Tabletoy and others. This will also allow us to evaluate further enhancements to the cellular programming model.

Appendix B

Implementing Applications on a Cellular Architecture - the Mandelbrot-set.

This was a presentation give to the University of Hertfordshire, upon the author's return from CAPSL, introducing DIMES and the work that was done.

Authors: Jason M^{c}Guiness[1,2], Colin Egan[2], Guang Gao[1].

[1]University of Delaware, Newark, DE.

[2]University of Hertfordshire, Hatfield, Hertfordshire, U.K. AL10 9AB.

mcguines@capsl.udel.edu

c.egan@herts.ac.uk

ggao@ee.udel.edu

B.1 Overview:

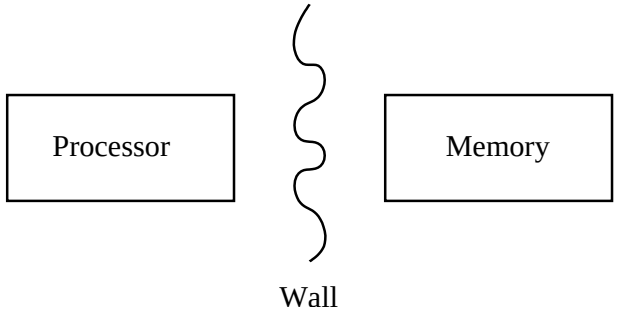
Recap from last week:

- The memory wall and cellular architectures: a solution?
- Programming models on Cellular Architectures, followed by a brief overview of Cyclops and DIMES/P-2.

New this week:

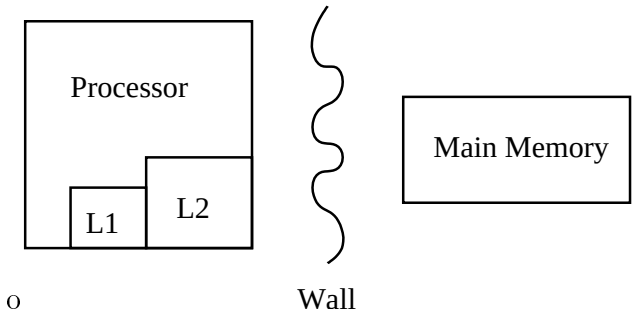
- An introduction to the Mandelbrot set.
- Threading and Work-Stealing applied to the Mandelbrot set.
- The programming implementation with regard to DIMES/P-2 and the execution details of the Mandelbrot-set application. *PLUS A LIVE DEMONSTRATION OF THE PROGRAM!!!*
- Latest work: Global Updates Per Second (GUPS) benchmarks.
- Conclusions & Future Work.

B.2 A recap on the memory wall. Part I: The processor viewpoint.



- Higher performance may be achieved through ILP by MII and/or pipelining. Various techniques are used to implement these goals, e.g. Register Renaming, Out-of-order instruction issue/execution, Branch Prediction, dynamic instruction scheduling, Value Prediction, Instruction Reuse, etc.
- But this causes a bottle-neck - upon a miss the recovery cost becomes increasingly high, because the memory cannot keep up with the required fetch rate.
- This leads to attempts to improve the performance of the memory.

B.3 A recap on the memory wall. Part II: The memory viewpoint.



- Increasing the levels of memory in the hierarchy, by placing levels of caches between the main memory and the CPU (or on the CPU).
- This reduces the memory wall, but on a cache miss the penalty is more severe. (Also this does not reduce the memory sub-system latency for an initial access, only upon subsequent access.)
- In both cases:
  - The hardware complexity and cost is increased.
  - The rewards obtained are balanced against known disadvantages.

B.4 The memory wall and cellular architectures: a solution?

- Why not place the processor in the memory, e.g. PIM architectures? Does this *remove* the memory wall?
- In principle due to the proximity of the execution units to the memory cells, the latency and bandwidth should be reduced.
- But due to the mixture of logic units on the silicone die, the gate density is reduced.

- To maintain gate density, more simple execution cores are used, such as RISC pipelines which may also omit branch prediction, for example.
- Thus the memory density and execution unit throughput are reduced. How may this be countered?
  - With the addition of a network interface to interconnect between the PIM chips. Thus each chip becomes a *cell.*
  - Thus reduced individual performance may be countered by interconnecting many of these cells together to build up a *cellular architecture,* e.g. Cyclops developed at IBM, Gilgamesh at NASA and Shamrock at Notre Dam.

B.5 Programming models on Cellular Architectures.

Cellular architectures have particular features that mean that their programming model is different to super-scalar processors:

- They have large (millions) of execution (or in cellular architectures *thread units*) which are simple.
- Memory access is irregular: Some memory is very close, thus fast, the rest is off-chip, so much slower.

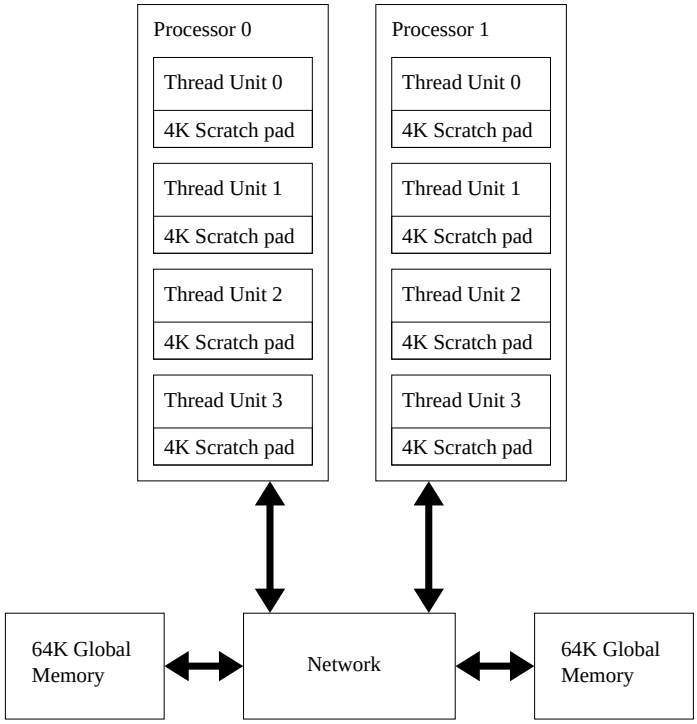
Research into appropriate programming models is on-going, the current model is pthread, but future directions include:

- For example thread percolation as a technique to perform dynamic load-balancing.
- In cellular architectures, multiple threads perform memory accesses independently. As a result of this, the memory subsystem could have some form of access model that allows these memory references to be effectively served. For example, the use of the location-consistency model could be used as a memory access model.

B.6 A brief overview of Cyclops and DIMES/P-2.

At the University of Delaware the first hardware simulation of a cellular architecture has been built under Hiro Sakane's group:

- This is called DIMES/P-2.
- It is a simplified implementation of the 32-bit CyclopsE design, one of the family of Cyclops architectures developed at the IBM T.J. Watson Research Center.



B.7 An introduction to the Mandelbrot set.

The Mandelbrot set is a *fractal* named after Professor B.B. Mandelbrot, who discovered the set in the 1960s. It is intimately related to the Julia set, also a *fractal*, discovered in the 1910s.

Both the Mandelbrot and Julia sets may be created by iteration of a very simple equation:

$$z_{n+1} = z_n^2 + c \tag{B.7.1}$$

In this equation, z_n is a complex number, where $z_0 = 0$. c is also a complex number, which is initialized to a value constant throughout the iterations. The iteration of equation terminates when:

1. Either n reaches the so-called "maximum iteration" value, m, a fixed constant, greater than zero.
2. Or $\mid z_n \mid$ exceeds the so-called "bailout" value, a fixed constant, usually set to the real value 4, for efficiency reasons.

B.8 The classic algorithm used to generate the Mandelbrot set:

1. Set the value of m, the maximum iterations, greater than zero.
2. Select a point from the complex plane, and set c to that value.
3. Initialize $n = 0$, $z_0 = 0$.
4. Execute equation B.7.1.
5. Increment n.
6. If $\mid z_n \mid \geq 2$ then that c is not in the set of points which comprise the Mandelbrot set. Go to 2.
7. If $n > m$ then that c is in the Mandelbrot set, i.e. $c \subset M$. Go to 2.
8. Go to 4.

B.9 Threading applied to the Mandelbrot set.

An overview of threading the Mandelbrot-set generation algorithm:

- An important property of algorithm to generate the Mandelbrot set is that the classification of each c in the complex plane is independent of the classification of any other c. Thus the Mandelbrot set may be implemented as a massively parallel application, thus potentially suited to cellular architectures. Indeed the Mandelbrot has has been used in as a benchmark for different architectures, such as fine-grain threaded-architectures and NUMA architectures.

The complex plane is divided into a series of horizontal strips. These strips may be calculated or rendered independently of each other, using separate render threads, as the classification of the points c within each strip is independent of such classification on other render threads. Therefore each render thread implements a slightly modified version of the classic algorithm, which is given in the threaded algorithm, given next.

B.10 The Render-Thread Algorithm.

1. The algorithm:
   (a) Set the value of m, the maximum iterations, greater than zero. Set the estimated completion-time, t, to ∞.
   (b) Set $c = x$, where x is the top-left of the strip to be rendered.
   (c) Initialise $n = 0$, $z_0 = 0$.
      i. Execute equation B.7.1.
      ii. Increment n.
      iii. If $\mid z_n \mid \geq 2$ then that c is not in the set of points which comprise the Mandelbrot set. Go to 4.
      iv. If $n > m$ then that c is in the Mandelbrot set, i.e. $c \subset M$. Go to 4.
      v. Go to 3a.

(d) Increment the real part of c. If the real part of c is less than the width of the strip to be rendered, go to 3.

(e) Calculate the average of t and the time it took to render that line.

(f) Set the real part of c to the left-hand of the strip. Increment the complex part of c. If the complex part of c is less than the height of the strip, go to 3.

(g) Signal work completed, set $t = 0$ (thus this thread is guaranteed not to be selected by the work-stealing algorithm).

(h) Suspend.

A load-balancing algorithm was added to move uncompleted work to threads that have completed their assigned work. This is because each strip will take a different amount of time to render.

B.11 The Work-Stealing Algorithm.

1. Monitor render threads for a work-completed signal. That thread that completes we shall denote as T_c.

2. Find that render thread with the longest estimated completion-time, t, note that each render thread updates this time upon completion of a line. Call this thread T_l.

3. Stop T_l when it completes the current line it is rendering.

4. Split the remaining work to be done in the strip equally between the two render threads T_c and T_l.

5. Restart the render threads T_c and T_l.

6. Go to 1.

This is a dynamic-programming solution to the load-balancing problem of work distribution between the render threads. Due to the selection of the slowest render thread, this algorithm may been see to be optimal. The author believes that this is an original application of work-stealing to Mandelbrot-set generation.

B.12 A Discussion of the Work-Stealing Algorithm.

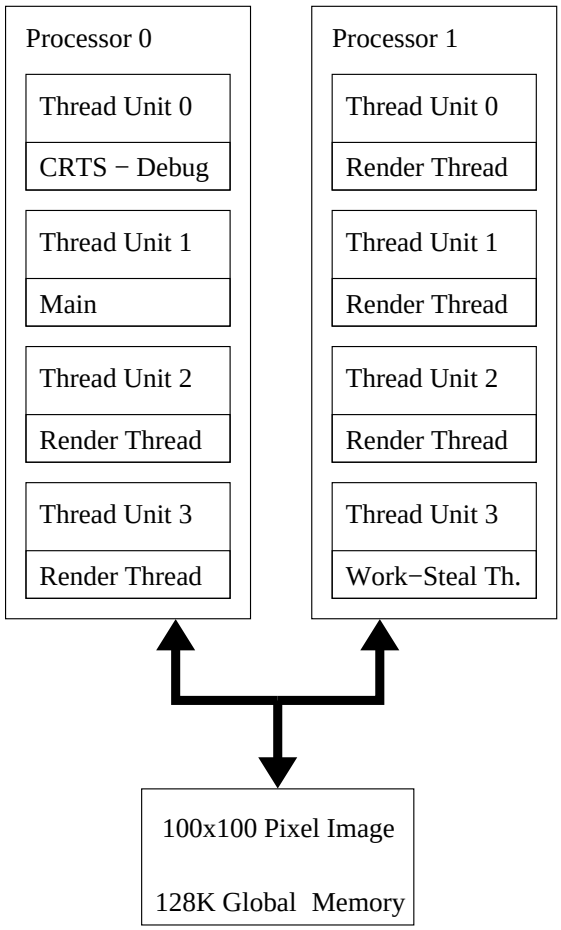
- The bandwidth of the single thread that implements that algorithm is the limiting factor in it's ability to scale.

- It is possible to scale this work-stealing algorithm, if one observes that the work-stealing algorithm operates upon a slice of the complex plane. This clue demonstrates that the work-stealing algorithm is recursive.

- Conversely this algorithm is able to tolerate failures in render threads. If a render thread stops responding, eventually it will be the slowest, unfinished render thread, and it's work will be stolen.If robustness is not required, then the image generated may be viewed as an array values. Each of these values is the classification of c. Thus if one has $p_{0...q}$ threads, each p_n thread initially classifies a point in the array offset by n, and once completed, moves along the array using a stride of q.

- This allows the use of a number of threads that is bounded by the number of points within the image. As this may be for an image of resolution 100×100, thus 10,000 points, this maps well on to cellular architectures.

B.13 The static layout of the render and work-stealing threads within the DIMES/P-2 system is shown below:



B.14 Execution Details of the Mandelbrot-set application.

B.15 Supercomputing Benchmarks: Global Updates Per Second (GUPS).

The GUPS benchmark is a very simple program that is effectively a cross-section bandwidth benchmark. It makes a large number of random updates to a large array:

Figure B.1: The image generated shortly after program start-up.

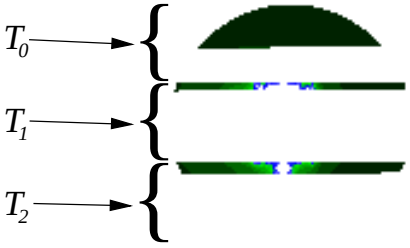


Figure B.2: Image generation has progressed, shortly before a work-stealing event.

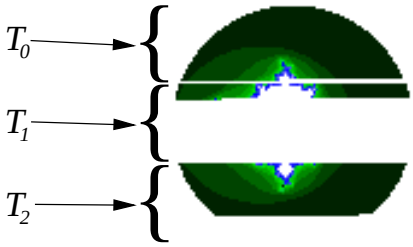


Figure B.3: Just after the first work-stealing operation.

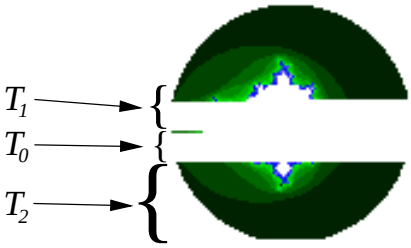
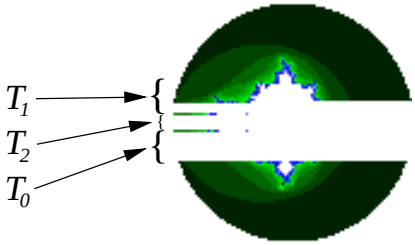


Figure B.4: The second work-stealing operation.



- for (i = 0; i < 30000000; ++i) table[random_integer] += random_value;
- This is complicated because for Cyclops we wish to perform this operation on the table across multiple processors.
- The limiting factor for the program is the memory access time due to the random reads & writes this confounds architectural features that may attempt to improve memory performance.
- *But* we are allowed to have a 0.1% errors in the table at the end of the benchmark.

This error rate is vital as it allows us to relax the locking used to access the global table. This relaxation means that updates to the table may not be done in sequential program order, thus introducing errors.

B.16 GUPS and DIMES.

Currently there have been three simple, initial implementations of this program, all run on DIMES/P-2:

1. A sequential implementation, with no threading.
2. Multi-threaded implementations:
   (a) Full locking on the table access, thus giving a zero error rate.
   (b) No locking at all on the table access, thus sacrificing error rate for speed. The error rate is as yet unmeasured, but this appears to run 10 times faster than the fully locked version above.

The justification for implementing GUPS with no locking is a statistical one. As the amount of memory on Cyclops is 1Gb/chip, with only 320 thread units/chip, then the likelihood of any two thread units accessing any one memory location at the same time is very low, much lower than 0.001 (our permissible error rate).

B.17 Limitations of current GUPS & DIMES.

- The pthread programming model is too simplistic:
  - It does not reflect the memory hierarchy. More sophisticated memory models (such as location consistency) will be needed to aid the programmer in effectively lay out the global table to make the memory accesses faster.
  - It does not directly support data or thread percolation.
- The DIMES/P-2 hardware has too few resources (only 8 thread units and 128Kb RAM) to be a realistic platform upon which to run these more sophisticated benchmarks.

B.18 Conclusion & Future Work.

- The Mandelbrot set is an ideal program to demonstrate and test massively parallel architectures, such as cellular architectures.

- The current run-time system and pthread programming model, although simple, is sufficiently powerful only for a certain sub-set of sophisticated applications.

- Future development of the Cyclops architecture towards Cyclops-64, with the development of DIMES/P-8 with more hardware resources (at least 32 thread units and 512Kb RAM) will allow the development and testing of more sophisticated programs, such as supercomputing benchmarks. These benchmarks and the greater hardware resources will allow further experimentation with the sophisticated programming models that have been suggested, such as thread percolation and location consistency.

Appendix C

The Challenges of Efficient Code-Generation for Massively Parallel Architectures.

This is copy of a conference paper submitted and accepted for the 11th Asia-Pacific Computer Systems Architecture Conference.

Jason M M^{c}Guiness[1], Colin Egan[1], Bruce Christianson[1] and Guang Gao[2].

Department of Compiler Technology and Computer Architecture, University of Hertfordshire, Hatfield, Hertfordshire, U.K. AL10 9AB. c.egan@herts.ac.uk[1]

CAPSL, University of Delaware, Delaware, U.S.A. g.gao@capsl.udel.edu[2]

C.1 Abstract

Overcoming the memory wall [121] may be achieved by increasing the bandwidth and reducing the latency of the processor to memory connection, for example by implementing Cellular architectures, such as the IBM Cyclops. Such massively parallel architectures have sophisticated memory models. In this paper we used DIMES (the Delaware Iterative Multiprocessor Emulation System), developed by CAPSL at the University of Delaware, as a hardware evaluation tool for cellular architectures. The authors contend that there is an open question regarding the potential, ideal approach to parallelism from the programmer's perspective. For

example, at language-level such as UPC or HPF, or using trace-scheduling, or at a library-level, for example OpenMP or POSIX-threads. To investigate this, we have chosen to use a threaded Mandelbrot-set generator with a work-stealing algorithm to evaluate the DIMES *cthread* programming model for writing a simple multi-threaded program.

C.2 Introduction

Integrating the processing logic and memory [21], termed PIM, is an approach to overcome the memory wall [121]. PIM architectures may improve both data-processing and data-access times, but the combined processor speed and the amount of memory may be reduced [21]. This may be overcome by connecting multiple, independent PIM cells, giving a *cellular architecture*. In this organisation, every thread unit is an independent single-issue, in-order processor, thus able to potentially access memory independently. Moreover, the different memory hierarchies may have different access timings and consistency models such as *location consistency* [40]. This gives rise to a number of code-generation problems, centred around the fact that to provide computational power, these systems are not only massively parallel, but have complex memory hierarchies.

Research also proceeded towards thread-generating compilers, for example, HPF and UPC [45], IBM XL Fortran and Visual Age C/C++, largely based upon OpenMP, all of which have their compromises. Some of these also have support for the various memory models.

Unfortunately general-purpose languages have been slow to adopt a sophisticated abstraction of the machine model, library-based approaches have developed, for example, the various implementations of OpenMP. But, the authors contend that library-based solutions to threading are too dependent upon the programmer to use effectively. For example, the explicit use of locks in programs is prone to error, with deadlocks and race-conditions that are hard to track down easily, introduced, even on systems with only a few processors. The development of suitable tools to debug multi-threaded applications has also been slow. Debuggers are in development, for

example for Cyclops [42], but there have been too few, with limited functionality.

As identifying parallelism both correctly and efficiently is very hard for the programmer to do, the authors contend that they should not do it. The compiler, equipped via these libraries with a detailed machine-model, could be able to use the programmer-identified parallelize-able variables and functions, to generate more efficient code. The authors identified little work investigating the software aspect of the code-generation problem for massively-parallel architectures. Unfortunately, if this case would continue, this shortcoming could adversely affect the popularity of such systems and maintain the perception that massively parallel architectures are too specialised and thus too expensive to be of more general use. Given the popularity of introducing multi-core processors, this position is set to become even more untenable.

C.3 Related Work

C.3.1 The Programming Models: from Compiler to Libraries

With such compute bandwidth, and parallelism, a number of problems for the programmer have been raised, primarily these are focused on the problems of memory reads and writes. Super-scalar chips have had mechanisms to hide these problems from the programmer, but the cellular architectures of such chips as picoChip [33] and IBM BlueGene/C [4] do not. Thus the programmer needs to know how memory reads and writes interact with:

- the software-controlled data-cache attached to that pipeline,
- the software-controlled data-cache of other on-chip pipelines,
- any global on-chip memory,
- the software controlled data-caches of other off-chip pipelines,
- the global on-chip memory that is on any other chips,
- any global memory that is not on any chip

- and finally, given the massive parallelism available, how to make efficient use of it.

For a programmer, the memory access models are important to understand, or to have a library or compiler that hides the details from the applications programmer. In the remainder of the paper the authors will focus on the IBM BlueGene/C architecture, and a prototype implementation of it called Cyclops [21, 30], that was implemented at CAPSL at the University of Delaware in collaboration with the University of Hertfordshire. The Cyclops architecture was prototyped in hardware, called DIMES/P, [91] which was used as the platform for executing the programming example, described later in this paper. In the following sections the memory access models will be discussed, leading on to a presentation of the authors' experience in developing a program for such an architecture. The experience gained from this will allow the authors to discuss the major problems that were faced, how, if at all, they were overcome, and the outstanding problem domains that, in the authors' experience, would hinder the acceptance of multi-core chips and, moreover such massively parallel designs as IBM BlueGene/C.

C.3.2 Programming Models on Cellular Architectures

The hardware differences between cellular and super-scalar architectures indicate that different programming models, to those used for super-scalar architectures, are required to make effective use of the cellular architectures [40, 42]. In the first two of those three papers, their authors propose the use of a combination of execution models and memory models, as already noted in this paper.

The primary concerns when programming DIMES/P, and thus any Cyclops-based architecture, were:

- How to manage the potentially large numbers of threads.
- How to easily express any parallelism within the input source-code.
- How to make correct, and most effective use, of the memory consistency models.

Some research has already been done regarding programming models for the threading, such as using thread percolation as a technique to perform dynamic load-balancing [62]. Another piece of research [22] investigated using multi-level scheduling-schemes: a work-stealing algorithm at the higher-level and a multi-threading technique at the lower-level to hide communication latencies. Alternatively there is research [88] into how to implement OpenMP efficiently on cellular architectures such as IBM BlueGene/C.

C.4 Programming for Cyclops - *cthreads*

This section will very briefly describe the *cthread* programming model, which is an early version of TNT [31, 42], then how it was used to implement the programming example, followed by a discussion of the implementation.

The implementation of the memory consistency models was relatively simple: earlier, unpublished, work on the GCC-based compiler had implemented a simple algorithm: all static variables were stored in on-chip memory, and the function call stack, including all automatic variables was placed in the scratch-pad memory.

As there was no language-level support for thread management, a library had to be implemented to support the thread management instructions in the Cyclops ISA, which was used as the basis for creating a higher-level C++ abstraction. This was because the cthread implementation, that closely followed a POSIX-Threads API, was considered far too primitive by the authors to be effectively used for programming Cyclops. This C++ API also included critical-section, mutex and event objects to allow for easier management of the lower-level objects.

To test these ideas, and the Cyclops architecture, a small, simple and embarrassingly parallel program to generate Mandelbrot sets [72] was created. In the following sections a brief overview of how this how this program may be implementation for DIMES/P.

Algorithm 4 The render-thread algorithm.

1. Set the value of m, the maximum iterations, greater than zero. Set the estimated completion-time, t, to ∞.
2. Set $c = x$, where x is the top-left of the strip to be rendered.
3. Initialise $n = 0$, $z_0 = 0$.
   (a) Execute $z_{n+1} = z_n^2 + c$.
   (b) Increment n.
   (c) If $\mid z_n \mid \geq 2$ then that c is not in the set of points which comprise the Mandelbrot set. Go to 4.
   (d) If $n > m$ then that c is in the Mandelbrot set, i.e. $c \subset M$. Go to 4.
   (e) Go to 3a.
4. Increment the real part of c. If the real part of c is less than the width of the strip to be rendered, go to 3.
5. Calculate the average of t and the time it took to render that line.
6. Set the real part of c to the left-hand of the strip. Increment the complex part of c. If the complex part of c is less than the height of the strip, go to 3.
7. Signal work completed, set $t = 0$ (thus this thread is guaranteed not to be selected by the work-stealing algorithm 5).
8. Suspend.

Algorithm 5 The work-stealing algorithm.

1. Monitor render threads for a work-completed signal. That thread that completes we shall denote as T_c.
2. Find that render thread with the longest estimated completion-time, t, note that each render thread updates this time upon completion of a line. Call this thread T_l.
3. Stop T_l when it completes the current line it is rendering.
4. Split the remaining work to be done in the strip equally between the two render threads T_c and T_l.
5. Restart the render threads T_c and T_l.
6. Go to 1.

C.4.1 Threading and the Mandelbrot Set

Due to the properties of DIMES/P, alternative techniques were not possible, as there are only 8 thread units between two processors. In this implementation, the complex plane was divided into a series of horizontal strips. Those strips may be calculated independently of each other, using separate threads, implemented as algorithm 4. However, each strip will, in general, take a different amount of time to complete, thus the threads would have completed their assigned portion of work at different times. Thus a work-stealing algorithm 5 performed the load-balancing between the threads.

The bandwidth of the work-stealing thread, algorithm 5, limited scaling to more worker threads, algorithm 4. But algorithm 5 would able to tolerate failures: if a worker thread stopped responding, its work would have been eventually stolen.

If robustness is not required, then the image generated may be viewed as an array values. Each of these values would be the classification of c. Thus if one has $p_{0...q}$ threads, each p_n thread initially classifies a point in the array offset by n, and once completed, would move along the array using a stride of q. This would allow the use of a number of threads that is bounded by the number of points within the image.

C.4.2 DIMES/P Implementation of the Mandelbrot-set application

In cthreads, each software thread was statically allocated to one of the 8 hardware thread-units in DIMES/P at program start-up. The software threads were:

1. The a thread was required for cthreads support and the debugger [42], if it were to be run.
2. The main loop of the Mandelbrot-set application.
3. The thread that executed the work-stealing algorithm 5. In principle, a worker thread could also run on this thread unit, but cthreads did not support virtual threads.
4. The remaining 5 threads were worker threads that executed algorithm 4.

Further details regarding the implementation may be found in [70].

C.5 Discussion

The limitations of DIMES/P prevented further study of the properties of this program: scalability and timings were not done because of the limited number of thread units (8) and memory capacity.

The memory model support, using the C/C++ keyword *static* by the compiler, made natural use of language-level syntax to map data into scratch-pad and on-chip memory made using these different memory models. The atomic, word-sized, memory-operations on Cyclops were not used for this problem, because of the multiple, read-modify-write operations that had to be maintained as an atomic unit. If the manual locking had been implemented within the compiler, then it may have been possible for the compiler to perform optimization on the locking of access to the data.

With regards to the thread library: in the opinion of the author's, the complexity of POSIX-Threads has been a hindrance to successful multi-thread program creation. Abstracting the algorithms that expressed the parallelism within the Mandelbrot program, for example the work-stealing algorithm, was not implemented for this paper, as this was considered to be potentially too closely coupled to the actual program in question. Ultimately this decision, in the authors' opinion, was flawed, and by extracting and abstracting the work-stealing algorithm from both the program and Cyclops, would have allowed a programmer to reuse that algorithm with other programs, thus separating the design of the parallelism from the details of the program that would wish to use it.

It is still an open question regarding what may be the ideal approach to parallelism: language-level support such as UPC, HPF or other language extensions, or within the compiler using trace-scheduling, or should it be at a library-level using, for example OpenMP or POSIX-Threads, or should it be within the architecture, such as the data-flow design. If programs more sophisticated than the one described in this paper are to be successfully written for these cellular architectures, then based upon this brief examination, it is the authors' contention that it would be highly advantageous to have:

- Compiler support for making use of any available the memory model of the architecture.
- Compiler support for locking, which would aid the programmer with writing code that avoids race-conditions.

- Reusable abstractions of techniques of implementing parallelism, such as work-stealing, or master-slave models. These abstractions could make use of both data and code locality to ensure that a thread unit re-executes the same code, if desirable.

The research presented in this paper is supported by the Engineering and Physical Research Council (EPSRC) grant number: GR/S58492/01.