An Integrated Functional Solution for Multi-core Programming on The Cell Broadband Engine

Nathaniel Azuelos

Department of Electrical & Computer Engineering
McGill University
Montreal, Canada

January 2009

A thesis submitted to McGill University in partial fulfillment of the requirements for the degree of Master of Engineering.

Abstract

Recent efforts in microprocessor development tend to the coexistence of several Central Processing Units(CPUs) on a single chip. The Cell Broadband Engine (CBE), the fruit of collaboration between Sony, Toshiba and IBM, integrates IBM's legacy PowerPC CPU with a new set of simple cores, all of which communicate through a high speed bus. The multiple cores on the CBE allow users to exploit the parallel nature of their programs. However, it is often difficult to efficiently extract the parallelism from an application and to distribute tasks in a suitable fashion. We propose a dataflow approach to CBE computing where the compiler is in charge of task partitioning and of the infrastructure for runtime distribution of tasks.

In this work, we present the NCC programming language, Squid compiler and runtime environment. NCC is a strict functional dataflow language that forces explicit variable dependencies, in order to exploit parallelism in the application. NCC code is thus written by the user without specifying parallelism explicitly. The Squid Compiler draws a virtual data flow graph from the NCC source. This graph is then partitionned according to implementation-specific criteria into tasks and supertasks. The individual tasks are then translated to ANSI C, and supertasks are analyzed and transformed into scheduling structures. All tasks are executed by the CBE's simple cores. The Squid Runtime Enviromnent (SRE) interacts with the generated scheduler to order tasks' execution, running the supertasks' scheduling, and managing garbage collection. The SRE runs on the CBE's PowerPC core as a separate thread to implement a host-device paradigm, and as resident code on the simple cores.

Résumé

Les récents efforts en développement de microprocesseurs tendent à une coexistence entre plusieurs Unités Centrales (UC) sur une seule puce. Le Cell Broadband Engine (CBE), le fruit d'une collaboration entre Sony, Toshiba et IBM, intègre le CU patrimonial d'IBM PowerPC avec un nouvel ensemble d'unités simples, communiquant entre elles avec un bus de haute vitesse. Les nombreueses unités présentes dans le CBE permettent aux utilisateurs d'exploiter la nature parallèle de leurs programmes. Cependant, il est souvent difficile d'extraire le parallelisme d'une application et de distribuer des tâches de façon appropriée. Nous proposons donc d'approcher la programmation du CBE sous une perspective de flux de données où le compilateur est chargé de partitionner les tâches et de l'infrastructure de la distribution dynamique des tâches.

Dans ce travail, nous présentons la langue de programmation NCC, le compilateur et l'environnement d'exécution Squid. NCC est un langage fonctionnel stricte de flux, qui force les dépendances entre variables á être explicites, afin d'exploiter le parallelisme d'une application. Le code NCC est donc rédigé par l'utilsateur sans spécifier le parallelisme explicitement. Le compilateur Squid dessine un graphe de flux de données virtuel issu du code NCC. Ce graphe est alors partitionné selon des critères particuliers à l'implémentation en tâches et supertâches. Chaque tâche est ensuite traduite en ANSI-C, et les supertâches sont analysées et transformées en structures d'ordonnançement. Toutes les tâches sont exécutées par les untiés simples du CBE. L'Environnement d'Exécution Squid (EES) interagit avec l'ordonnanceur généré pour ordonner l'exécution des tâches, commander l'ordonnancemenet des supertâches, et gérer la collection de poubelle. Le EES roule sur l'untié PowerPC du CBE en tant que fil

d’éxécution séparé afin d’implémenter un paradigme hôte-dispositif, et de code résident sur les untités simples.

Acknowledgements

I would like to thank my supervisors, Profs. Zilic and Gross, for giving me the opportunity to study this fascinating field, for their advice, insight, and numerous comments in the writing of this thesis. I especially want to thank Bojan Mihajlovic for his friendship, and for always being there to answer questions; an incredible help without which this work would have never come to fruition. David Becerra for the close collaboration and friendly rivalry. Further thanks to the “elders”, Jean-Samuel Chenard, Stephan Bourduas, Marc Boule, Atanu Chattopadhyay and Henry Chan for being such towering figures and true role models. A special thanks to the 4th, 5th and 6th floors for making lab life so enjoyable. Ashraf Haddad for our late night musings about politics and religion, as well as to all the other musers: Carmen, Marwan, Dani, Sadok, Mike, Francois and Euisoo. A personal thanks to my parents for their love and for the education they gave me, with its enormous emphasis on academic excellence. To my brother Ilan for his complicity and for always pushing me to sharpen my mind. To Olivier Rubel and David Tellouk for their friendships and laughs.

רבי אלעזר בן עזריה אומר, כל שחכמתו מרובה ממעשיו, למה הוא
דומה--לאילן שענפיו מרובין ושורשיו מועטין, והרוח באה ועוקרתו והופכתו
על פניו. וכל שמעשיו מרובין מחכמתו, למה הוא דומה--לאילן שענפיו מועטין
ושורשיו מרובין: אפילו כל הרוחות שבעולם באות ונושבות בו, אין מזיזות
אותו ממקומו.

(Talmud Avot 3, 21)

Contents

List of Figures

List of Tables

Chapter 1

Background: Multi-Core Architectures and the Cell Broadband Engine

The exponential growth in transistor density dictated by Moore's Law has been the primary source of increase in computing power in microprocessors since their inception. Microprocessors accelerated applications by exploiting instruction level parallelism present both statically and dynamically through advanced techniques and analysis, by compilers on the one hand, and intelligent architectures on the other. As integrated circuits further grew in size, an increasing proportion of their area was dedicated to multi-level caches in order to confront the high memory bandwidth consumed by data-intensive applications. These techniques were quickly subject to the law of diminishing returns. As transistor density further increases, multi-core processors quickly became an interesting alternative.

Contrary to on-chip multi-core processors, multi-chip multiprocessor systems have existed for a long time and were thouroughly studied. In most cases, the differences between implementations of multiprocessors lay in the method of communication and memory coherence. Intuitively, the problems and solutions developed for multiprocessor systems would apply as well to multi-core systems. From a high-level perspective, the solutions proposed were indeed similar. However, the physical proximity of the cores has led to subtle and interesting differences with traditional multiprocessor implementations. The communication latencies are drastically shorter, and cores may share caches and random access memory. The Cell Broadband Engine (CBE) is IBM's solution to the multi-core challenge. It consists of a heterogeneous mix of its legacy PowerPC CPU (PPU) and a varying number of Synergistic Processing Units (SPUs). It is generally considered one of the most computationally powerful CPUs on the market [1].

The long-awaited arrival of multi-core CPUs had an important impact on software development. Until recently, parallel programming was the domain of experts and the concern of a select few. With the advent of multi-core CPUs, software developers now need to consider parallel implementation of their code in order to increase performance for the common machine. Unfortunately, there are great difficulties in parallel software development, such as synchronization, code and data partitioning, and subtle hazards that may occur. In most cases, the parallel portions of an application are explicitly specified and synchronization methods have paramount importance. Further, intimate knowledge of the multiprocessor architecture at hand is necessary.

High-level design principles of multiprocessors have provided a basis to the hardware implementation of multi-core CPUs. Similarly, software paradigms applicable to the former were applied to the latter. Unfortunately, the difficulties of parallel programming for multiprocessors have also been inherited. However, the ubiquity of parallel systems propagated these difficulties to a much larger spectrum of programmers. Fortunately, alternative software paradigms developed for multiprocessors still apply in the multi-core era, and are well worth being revisited.

One such alternative solution is the fruit of a long history of research into programming languages. *Functional programming languages* describe the computations rather than dictate their order. The order of computation can be deduced from an analysis of the data dependencies between computation quanta. Further, parallel threads of computation become evident and can be scheduled to be performed simultaneously.

The order in which parallel threads extracted from functional code are executed can be better determined at runtime. It is extremely difficult for a compiler, and even more so for a programmer, to predict accurately the resource usage of an application at a given moment. Therefore, such work is best left to a *runtime environment*. This approach to functional programming is called *lazy*, as data is computed only once it is needed. There is a number of lazy programming languages, the most popular of which is Haskell. A large body of work has been produced on the subject of the implementation of lazy programming languages on multiprocessors.

This thesis suggests a three-step solution to the challenge of multi-core programming. The CBE is chosen as a target for this implementation as it is easily available and a good example of future CPUs. A functional programming language bearing much resemblance to C is described. Its compilation is then documented. Finally, the anatomy and mechanics of the runtime environment in charge of distributing computation are elaborated. However, an overview of multi-core architectures, parallel programming, current solutions, as well as the motivations for our work are first offered.

1.1 Multicore Architectures

The recent migration from uniprocessors to the multi-core era stems from the realization that uniprocessor development has reached fundamental limits. The three major concerns in the development of new architectures have been identified for some time, and are known as the three "walls". The first and foremost is the "Power Wall": increasing clock frequency, transistor count and die size increases power consumption density at an unsustainable rate. Serious considerations must be taken to reduce power density. Further, the increase in the size of silicon dies causes the distance to outside interfaces to increase and is designated as the "Memory Wall". The access time for memory loads and stores to Dynamic Random Access Memory (DRAM) is now attaining hundreds of clock cycles. The final challenge is related to the amount of instruction level parallelism (ILP), the "ILP Wall". Dedicating an increasing amount of resources to exploit parallelism between low-level instructions faces the law of diminishing returns.

Despite these drawbacks, Moore's law still applies to the size of integrated circuits. The large amount of die size available brings about new possibilities to tackle these problems. First, the more cores lay on a chip, the higher the theoretical maximal computing power. However, deciding the size and functionality of these cores is quite important. With its dual and quad-core processors, Intel has for now chosen the simplest way: replicating complex cores on a single die that share cache resources. It seems however that this approach is temporary and Intel has already created a prototype [2] with 80 simple cores on a single chip.

The use of simple cores has been subject of some study [3], and is proving popular. The CBE currently includes up to 8 simple Single Instruction Multiple Data (SIMD) cores on a single chip. Graphical Processing Units (GPUs) have integrated a large amount of cores for quite some time, and NVIDIA has recently provided application programming interfaces (APIs) to adapt their GPUs to general purpose computing. Sun's Niagara [4] T1 and T2 processors also consist of 8 cores with multi-threading capability and shared L2 cache. The popularity of the simple core approach is due to the following considerations [5]:

1. Power: eliminating ILP-exploiting units to the benefit of extra cores leads to a net benefit in computing power per Watt [6].

2. Cost: the cost of large chips is amortized by the increased computing power of new cores.

3. Defect Tolerance: among a large number of cores, the loss of a few to silicon defects does not void the usability of a chip. Sun and IBM deliver chips with some cores turned off to increase die yield in the Niagara and CBE respectively.

Kumar *et. al* [7] however argue in favor of the benefits of a heterogeneous architecture, where cores with different capabilities reside on the same chip. They propose to maintain at least one core to efficiently run scalar sequential code. Using this model, they achieve an increase in performance of up to 63%. Such a result is a consequence of Amdahl's law [8] that finds the effect of the sequential portion of a code in a parallel environment as an asymptotic limiting factor. The issue of homogeneous *v.s.* heterogeneous multiprocessor architectures was studied by Andrews [9]. It led to simulations for multi-core systems by Becchi [10] and Sondag [11] that show that significant runtime performance can be exploited from a dynamic profiling of threads during execution in heterogeneous systems.

1.2 CBE Architecture

The Cell Broadband Engine (CBE) [12] is IBM and Sony's solution to the multi-core challenge. The CBE was initially distributed at very low cost ($400) on Sony's Playstation 3 gaming platform, and is now an integral part of the US Department of Energy Road Runner super-computer project that claims to have achieved the elusive PetaFLOP. The work presented in this thesis attempts to present an integrated solution to the goal of a new and efficient platform to multi-core programming. We focus our work on an implementation for the CBE, due in part to its popularity, but mostly because we believe it provides a realistic preview of technologies to come.

The power of the CBE does not rest solely on the number of its cores (7-9 on current platforms). It incorporates a single unit of IBM's legacy PowerPC computing core (PPU) with vector extensions and 512KB of cache. Its other cores are SIMD cores with 128-bit register files designated as Synergistic Processing Units (SPUs). These are "stripped": no hardware units are included to perform out-of-order instruction execution, predictive branching and other such ILP-exploiting features. Finally, its Element Interconnect Bus (EIB), a 4-ring packet network, allows extremely fast communication between cores and with external interfaces. Figure 1.1 shows a top-level view of the CBE.

In opposition to the PPU and most other cores in different architectures, the SPU does not have a cache. It accesses a small 256KB *local store* that allows very fast local memory access and is **not** coherent to Main Memory. To access Main Memory, Direct Memory Access (DMA) commands are provided. The CBE is not designed as a strictly distributed memory

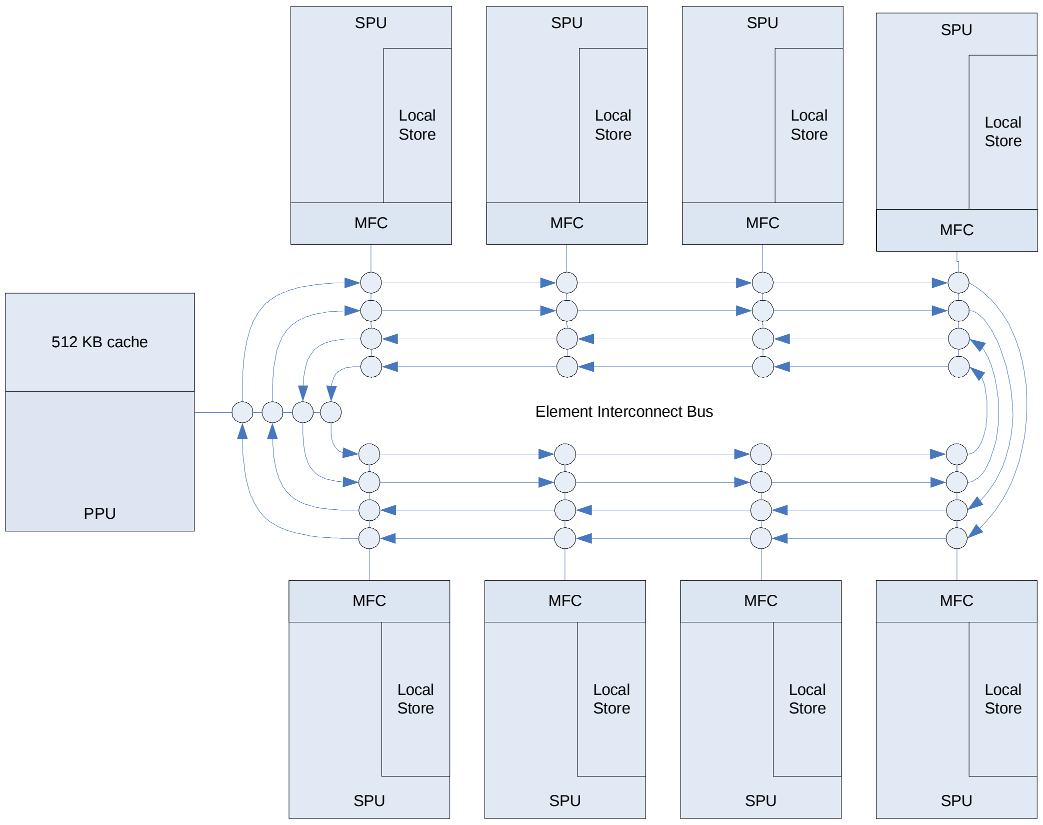

Fig. 1.1: A top-level view of the Cell Broadband Engine

machine however; every local store is mapped to locations in the global virtual memory. The CBE designers thus allowed programmers to use both message-passing and shared memory approaches.

1.2.1 The PPU

The PPU implements exactly the PowerPC standard, version 2.02 [13]. Its virtual memory is designed for 64-bit addressing, although it still supports 32-bit addressing for backward compatibility. In fact, it is sometimes preferable to use 32-bit addressing, when atomic or mailbox messages (described below) of memory addresses are involved. The PPU's VMX extension offers it SIMD capability, although its smaller register size does not allow it to match the computational power of the SPUs.

1.2.2 The SPUs

The SPU architecture is that of a simple two pipeline SIMD processor. The simplicity of the design causes many low-level code generation considerations. First, the two pipeline design allows the simultaneous execution of two instructions. The first pipeline is dedicated to arithmetic operations, whereas the second to memory accesses and shuffle instructions. The second, more significant, consideration is the fully SIMD nature of the processor. Contrary to the PPU, the instruction set of which is augmented by SIMD capabilities, the SPU architecture *only* performs SIMD instructions. In particular, all memory accesses are aligned on a 16-byte boundary. Consequently, to execute scalar code, re-alignment operations are very often needed, particularly when accessing array elements. Scalar code is thus very costly in SPU performance.

Whereas scalar code is very inefficient on the SPU, parallel code is very efficiently processed. The 16-byte registers allow 16 simultaneous operations for `char` data types, 8 for `short`, 4 for `int` and `float`. This capability allows acceleration by a significant factor, although the inevitability of scalar code usually hinders performance.

The small amount of memory in local store is an important limiting factor when using SPUs. The local store must hold stack, heap, and code at runtime, and the optimization of the space used is a challenging problem. Specifically, this problem can be limiting in the amount of code available. Libraries can be quite large, and space devoted to code grows too quickly with respect to the space left for processing data. *Code overlay* is an important technique in SPU programming: dedicated regions of memory are reserved for a group of code elements. Code elements can be swapped so only one of a group may be resident on the local store at any time, significantly saving space and extending SPU capability.

1.2.3 Communication

The EIB [14] can reach a theoretical bandwidth of over 300 GB/s, although practically, only about 200GB/s is realizable. The ring network transfers packets in sizes of 16 bytes. As a result, data transfer sizes must be multiples of 16 bytes, although one, two, four and eight byte packets may be sent. Further, when accessing memory outside local stores, a data alignment of 128 bytes is required of the hardware. This requirement is due to the fact that the PPU cache performs snooping on bus addresses. Since cache lines are 128 bytes wide, the memory addresses snooped must be so aligned. In order to initiate transfers to and from SPEs, it accesses the target SPE's Memory Flow Controller (MFC) through its address in the global virtual memory. These memory transfers are dubbed *proxy DMA* commands, and are handled slightly differently than SPU-initiated transfers. To every core is attached a Memory Flow Controller (MFC) to interface access to the bus. The MFC can read and write to the local store, as well as perform atomic operations. All instructions used by SPUs to send commands to the MFC run through the SPU's memory pipeline. Communication with the MFC is done through the following channels:

- SPU event channels, to control the response to external events
- SPU signal notification channels, used by MFCs to send 32-bit messages to one another
- SPU decrementer channels, for timing purposes
- MFC multisource synchronization channels, to enforce barriers in communication
- SPU and MFC read mask channels, to access specific elements of MFC and SPU control registers
- SPU state management channels, used by the Operating System
- MFC command channels, to initiate, coordinate and act upon data transfers
- MFC tag status channels, to identify the status of specific transfers
- MFC mailbox channels, that offer a more elaborate short message communication mechanism

These facilities provide the MFC an array of mechanisms, some simple and others complex, to communicate with other MFCs. However, most of these are too complex to be used by the common user or a robust runtime environment. Of particular interest to the application developer are the following: DMA transfers, atomic operations, and mailbox communication.

1.3 Using MFC Facilities

DMA transfers, as previously stated, are guaranteed to be memory coherent: data in main memory, PPU cache and data being distributed through the MFC, are identical for a given memory address. Hence, simultaneous operations on memory locations will not occur. In order to optimize the use of the EIB, the MFC may reorder memory transfers, and the commands sent to the MFC, although coherent, may cause data hazards with respect to the application's purpose. Further, in order to avoid long delays due to data transfers, the SPU overlaps data transfer and execution. Synchronization mechanisms are thus required to ensure program correctness. Tag group status registers, *fences* and *barriers* are such mechanisms.

Tag Groups are particularly useful in CBE programming. At several moments during an application lifetime arises the need to synchronize execution and the completion of a data transfer. Further, several DMA transfers may be active simultaneously, and there might be a need to synchronize on one's completion rather than every single one. To that effect, to every DMA transfer is associated a tag group identification, ranging from 0 to 31. This identification mechanism allows developers to synchronize multiple DMA commands with the same tag, independently of all others. Tag mask registers are present to specify, upon synchronization, which groups status must be checked.

Fences and barriers offer synchronization between data transfers rather than with program execution. A fenced MFC command forces the MFC to perform the data transfer only once all previously issued commands with the same tag group have completed. Barrier commands force all previously issued commands, no matter what their tag group, to not follow it, and no command issued thereafter may precede it.

Another important tool made available by the architecture is the *DMA list*. The DMA list is the CBE's tool to perform *gather* operations; data scattered in memory is collected in a continuous block of memory. It offers software the possibility of avoiding the specification of a single source, single destination for every DMA transfer. A single DMA command is issued, where a list of transfers is sent to the MFC. Every transfer is identified by its size and source address. A single destination is specified, to which all transfers will accumulate. DMA lists thus allow an efficient way of performing several transfers at once. It is interesting to note that there is no equivalent *scatter* command made available by the architecture.

Beyond DMA transfers, atomic operations allow simple and efficient coherent arithmetic and memory access to memory, whether local or foreign. A very small atomic command cache makes their use particularly attractive for synchronization purposes. Mailbox messages are well suited to that purpose as well. They are not used in this work however, and we shall thus refrain from elaborating on their details.

1.4 Multiple Buffering

Tag groups enables the use of a very important technique in CBE programming: *multiple buffering*. It is used generally in the context of large independent parallel loops, where the data access pattern is well determined in advance. A fixed, limited number of input and output buffers are created in SPU local store to overlap one loop's data transfers and the next's communication. By using differnet tag groups, synchronization may be performed on one iteration's fetch while another's continues still. Meanwhile, one output buffer's writeback and another input buffer's fetch may be assigned the same tag group, in order to validate the execution of an iteration dedicated to these buffers. The following code offers a double buffering example, and is represented in Figure 1.2:

```
//create 2 input and output buffers
float input_buf[2][256];
float output_buf[2][128];
```

```
int mm_address_in = ...;
int mm_address_out = ...;

mfc_get(input_buf[0], mm_adress_in, 0); //initiate first transfer

int i;
for(i=0; i<C; i+=2){
    mm_address_in += 256;
    mfc_get(input_buf[1], mm_adress_in, 1); // get next data for buffer 1

    // ensure last get and put are complete
    sync_tag(0);

    // compute on buffers 0

    mfc_put(input_buf[0], mm_address_out, 0); // write back buffer 0 result
    mm_address_out += 128;

    mm_address_in += 256;
    mfc_get(input_buf[0], mm_address_in, 0); // get next data for buffer 0

    // ensure last get and put are complete
    sync_tag(1);

    //compute on buffers 1

    mfc_put(input_buf[1], mm_address_out, 1); // write back buffer 1 result
    mm_address_out += 128;
}
...
```

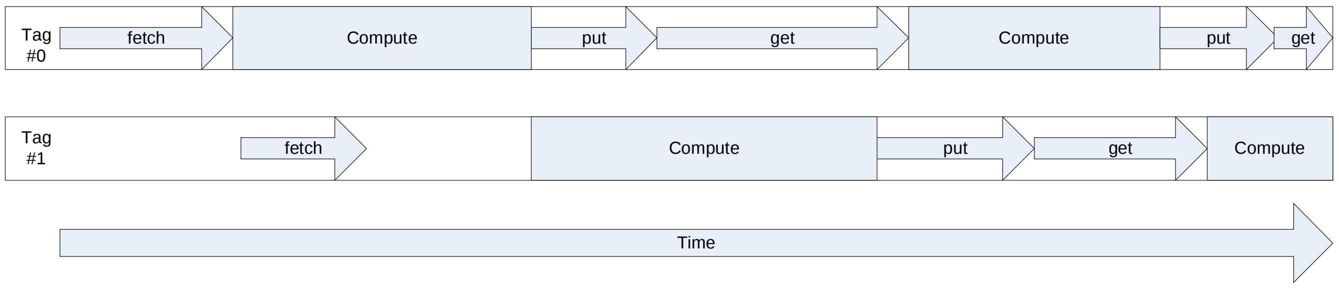

Fig. 1.2: Double buffering using tag groups

1.5 Programming for the CBE

The heterogeneity of the CBE's cores and the multiple details in its inter-core communication mechanisms add further complexity to the parallel programming problems described previously [15]. The CBE Programming Tutorial [16] and Handbook [17] offer several programming techniques and some others have been offered by the literature [18, 19]. DeKruijf proposes the use of Google's MapReduce algorithm to transparently distribute tasks to SPUs.

As it stands, `gcc` implements several optimization techniques, notably auto-SIMD-ization of SPU code. A powerful SDK including a full-system simulator, as well as a runtime API are provided by IBM/Sony. A set of macro commands are available in the API to simplify channel writing to the MFC [20]. Unfortunately, `gcc` does not perform code partitioning or implement multiple buffering techniques. Beyond `gcc`, several solutions have been developed to simplify code generation for the CBE. These usually follow the traditional parallel programming approaches, as both MPI [21] and OpenMP [19, 22] tools are available, the latter as part of IBM's XL compiler suite. An implementation of the Java Virtual Machine has also been developed as an academic project [23].

IBM's Accelerated Library Framework (ALF) [24] offers programmers an integrated environment to partition computational kernels written by the user. Divided tasks, or *work blocks*, are dynamically distributed by a runtime environment to which the main threads passes tasks to accelerate. However, it does not do so from a simple source code, and kernels must be defined as such by the user. To respect correct execution order, dependencies between tasks must be specified explicitly.

Contrary to ALF, Zhao and Kennedy [25] have developed a dependence-based compiler for Fortran targeting the CBE. They use Allen and Kennedy's [26] *dependence matrices* to study dependencies in loop nests to optimally divide the code. They use a static model of task distribution based on a *fork-join* model of task spawning.

Beyond compiler technology, APIs were developed or ported to accelerate applications on the

CBE. In particular, an SPU implementation of the Basic Linear Algebra Subprograms API [27] is distributed by IBM to enhance the programming of scientific applications. FFTW [28] has also offered, in its latest version, support for computation on Cell [29] in its latest framework.

1.5.1 Scheduling

Just as for any other parallel system, programming for the CBE involves a choice of runtime model. Statically allocated task distribution models are most common, and fit well with OpenMP and MPI. However, Blagojevic *et. al* show that dynamic adaptations can increase performance. Three Dynamic task distribution mechanisms were studied [30, 31].

The first mechanism presented by Blagojevic introduces task-level parallelism, by allowing fair sharing mechanisms of MPI SPU processes. The second attempts to evaluate the amount of loop partitioning necessary to optimize SPU use. Finally, the third studies the throughput of SPUs to throttle the granularity level of MPI tasks, thereby avoiding local maxima. In spite of the overhead one may expect from such an approach, it proves to be of little consequence, and good performance is achieved. Unfortunately, this appcoach is restricted to very large, completely regular applications, and does not apply to irregular tasks. It shows, however, the potential of dynamic adaptability in CBE task distribution.

Chapter 2

Parallel Programming and Functional Languages

Long before the age of multi-core processors, researchers attempted to accelerate large, complicated problems using several computers computing different aspects of the problem. The problems encountered then are fundamentally the same as those parallel programmers face now. There are several hazards implicit to computing in parallel. For instance, several actors may share physical resources such as memory, disks, etc. and require OS support. Yet the biggest difficulties lay in data coherence; if several threads must share a common data element, the order in which reads and writes occur could have a corrupting effect on the result computed. Synchronization is also an important problem: a thread should not use data created by another thread if that data's processing is not complete.

Many of the solutions to these problems tend to be solved through some amount of hardware support. Atomic primitives allow read and write operations given memory locations that do not overlap to cause unexpected behavior. However, many of the issues are also directly dependent on the problem posed, and require software solutions. Coherent access to data must respect dependencies in the data access patterns that the application requires. Surreptitious problems such as *deadlock* can occur as well, when several resources depend on one another to advance execution, causing none to do so.

2.1 Parallel Programming Design

Given these challenges, integrated solutions to parallel programming have been developed [32]. The first such solution offered was High Performance Fortran [33] (1993). Ulterior solutions fundamentally differed on the memory model used; processing elements (PEs) may share a global memory space or not. In the first case, much care must be taken to guard against data hazards and to ensure proper synchronization. OpenMP [34] is the most widely used language extension in shared-memory parallel systems architecture. It provides a set of preprocessor directives to indicate to compilers the method by which they are to divide single-source code into threads and enforce synchronization.

In the alternate model, every PE operates independently. Parallelism and synchronization is achieved by means of the underlying communication architecture of the system. PEs thus communicate through *message passing*. In the popular Message Passing Interface (MPI) [35], library routines are provided to offer developers an abstraction of the low-level communication mechanism. A certain advantage of the distributed memory model is the modularity and scalability it provides to parallel system architects. Indeed, if library routines can adapt to different systems, there is little added complexity to adding hardware.

These two solutions are used as extensions to existing programming languages commonly used in High Performance Computing (HPC). However, these and others impose difficult decisions on developers. If an application is to be parallelized, it must be divided in moderately independent threads of computation. If an excessive amount of synchronization exists between threads, a large amount of time will be lost as threads wait for one another. Further, any communication incurs an associated overhead. The decisions taken in the division of code are thus quite consequential to performance.

As a natural evolution from machine-level code where instructions are performed in a sequential way, current programming languages dictate, step by step, the order in which individual quantas of computation must be executed. The extension of such languages becomes naturally awkward as the number of concurrent processes increase; it is quite difficult to transform a sequential approach into a parallel one. Some studies on the efficiency of novice parallel programmers expose that current solutions are quite cumbersome and difficult to use, and have a rather steep learning curve [36, 37].

2.2 Task Partitioning

No matter what tool is used in parallelization, parallel code must be divided in separate units, as it must be run on separate processors. The paramount issue thus becomes the efficient and correct division of the code, and a proper analysis by a user or compiler is required

The parallelism present in an application can be exploited in several different ways. A code may have two independent and unrelated computations to perform, the results of which are required after a long period of time has elapsed. They can both be computed simultaneously and a blocking mechanism ensures both work units have completed to consume their results. A more common source of parallelism, however, is from loops. Indeed, typical programs use loops very frequently and at several levels of hierarchy. It occurs quite often that loop iterations are mutually independent, a property that offers a clear opportunity for parallel partitioning. Loop iterations with mutual dependencies also have the potential for parallel execution and deep mathematical analysis has developed several analyses of the problem, starting with Lamport [38].

The partitioning problem is a mathematical analysis of a set of data distributed along three axes, or *spaces* [39]:

1. the *iteration* space, or the uniprocessor equivalent of the time progression of a loop,
2. the *data space* representing the subset of array elements accessed
3. the *processor space* to which individual or groups of iterations are assigned.

The task partitioning problem can thus be restated as *the search for hazard avoiding concurrent accesses to data across the iteration space.*

A location on iteration space is generally denoted by the use of an *index* associated to the loop. In the simple example:

Example 1.

```
int X [20];
for(i=0; i<10; i++)
    X[i+5] = X[i];
```

the loop index is denoted by the variable `i`. The data space of the variable `X` is the subset of values indexed from 0 to 15 as all these locations will be read or written to. It can easily be seen that the loop can be divided into two parts. In the first five iterations, the element of `X` written to is not one of the original values of the array. However, the next five iterations are those that were accessed in the first part. The direction of the iteration axis must thus be respected in the computation of the loop.

The observation that led to the division of the loop in two parts is derived from the analysis of the array indexes accessed. In our example, both indexes are linear equations in `i`, and are said to be *affine* in `i`. An affine expression is defined as follows:

Definition 1. An expression is affine with respect to a loop index variable if the expression applies a linear operation on the variable; the variable may be multiplied by and added to constants or variables themselves affine with respect to the loop index.

The presence of affine indexing indicates that the data access pattern is regular across all iterations. It does not, however, indicate that the iteration space can be directly mapped to a processor space of equal size, as we have already observed. Indeed, the loop bounds in Example 1 cause a data hazard across the fifth iteration. A data dependence analysis across loop bounds must then be applied.

It is evident that the loop bounds have a direct impact on the presence of data dependencies across loop iterations. If the upper bound of `i` in our example was limited to 5, the loop would have been completely parallel. The impact of loop bounds is complicated further in the case of nested loops. These can be expressed by a set of linear inequalities, as shown in the following example devoid of data dependencies:

```
int X [20][100];
for(i=0; i<100; i++){
    for(j=i; j<10; j++){
        X[i][j] = 0
    }
}
```

Four inequalities may be written in this case:

$$\begin{aligned} i &\geq 0, \\ i < 100 \Leftrightarrow 100 - i &> 0, \\ j \geq i \Leftrightarrow j - i &\geq 0, \\ j < 10 \Leftrightarrow 10 - j &> 0 \end{aligned}$$

and can be simplified in the general case using *Fourier-Motzkin analysis* [40]. In general, nested loops may be formalized as:

$$\mathbf{Bi} + \mathbf{b} \geq \mathbf{0} \tag{2.1}$$

In our example:

$$\begin{bmatrix} 1 & 0 \\ -1 & 0 \\ -1 & 1 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} i \\ j \end{bmatrix} + \begin{bmatrix} 0 \\ 99 \\ 0 \\ 9 \end{bmatrix} \geq \mathbf{0}$$

Similarly, array references can be formalized as:

$$\mathbf{Fi} + \mathbf{f} \tag{2.2}$$

Let us determine what constitutes a data dependence. There are three possible cases:

1. **True Dependence**: a write operation is followed by a read. Inverting the order would cause the old rather than new value to be read.
2. **Antidependence**: a read is followed by a write, causing the opposite problem.
3. **Output Dependence**: two writes follow each other on the same memory location.

A data dependence is thus certain to be present if there is:

1. at least one write
2. both references are affine
3. the references respect the condition:

$$\mathbf{F}_1\mathbf{i}_1 + \mathbf{f}_1 = \mathbf{F}_2\mathbf{i}_2 + \mathbf{f}_2 \tag{2.3}$$

   where i_1 and i_2 are members of Z^{d_1} and Z^{d_2} respectively.

Equation 2.3 represents a set of *Diophantine equations* that is solvable through the Greatest Common Divisor (GCD) theorem:

Theorem 1. *A linear Diophantine equation:*

$$a_1x_1 + a_2x_2 + ... + a_nx_n = C$$

has an integer solution iff the greatest common divisor of $(a_1, a_2, ..., a_n)$ *divides C.*

Theorem 1 is of great use in compiler theory; compilers may study array references in order to perform efficient partitioning using an array of techniques. Functional languages such as Haskell also perform the test efficiently at runtime using *memoization* [41].

2.3 Functional languages

The difficulties of parallel programming were not lost on the HPC community in the past quarter-century. Extensive research has been done on the development of alternative programming languages. The most commonly used programming languages such as C/C++ and Java are termed *imperative languages.* These languages are typically a high-level abstraction of machine code. In the same way that microprocessors perform instructions one at a time, the high-level statements of imperative languages are expected to do so as well. The time progression of program execution is bound to the order in which statements are written.

As an alternative, functional languages were devised to work from a mathematical perspective. They originated with lambda calculus [42] and express computation as the relation between different functions. The mathematical rules of commutativity and associativity dictate the order (or disorder) in which different functions are evaluated. From this early model, several languages were developed.

The fundamental property of functional languages that allows compilers to evaluate code in different possible orders is **referential transparency**. This property states that the result of one expression does not change if it is executed at any time with the same set of inputs. An expression may thus be replaced by its equivalent value, or any other expression of equivalent value. As a consequence, functional code may not contain any form of state as is common in imperative languages. In an execution context, this allows a *runtime system* to decide on the appropriate timing for the evaluation of an expression at runtime. In parallel systems, different threads of execution may be computed in an order dependent on the availability of data and hardware resources.

The possibility of modifying the evaluation order of functional code has led to a schism in the functional programming paradigm. The computation of an expression can be delayed until the moment when the expression's value is required. The Haskell [43] programming language follows this *lazy* evaluation approach. It implements a "pull" model of evaluation that can lead the system to avoid wasteful computation. By contrast, *strict* evaluation performs all the computations described by the code and thus *eagerly* advances program execution. Eager evaluation removes much overhead, since variables can be passed by value rather than through the use of symbolic data constructs. Strict programming languages are typically simpler to implement for this reason and several popular languages were developed under this model such as LISP (1958), ML (1973) and CAML (1985).

2.4 Implementation of Functional Languages

The vast amount of flexibility offered by functional languages in a parallel context imposes several decisions on the following aspects of the compiler and runtime environment [44]:

- **Evaluation Model**: Code can be seen either as a set of functions to execute (graph reduction approach) or as the consumption of input data (dataflow approach).
- **Storage Management**: If the evaluation of expressions is distributed in time, so is the creation and consumption of data. Such dynamic management and intelligent garbage collection are paramount issues to a runtime environment.
- **Communication Model**: As threads are created dynamically, their placement may either cause the transfer of data or become a consequence of its physical location on a PE.
- **Load Distribution**: The assignment of a thread to a PE may be requested by it or forced unto it.

In the first evaluation model, functions are associated to data at runtime. A graph of operations thus needs to be reduced by applying these functions to others that may have yet to be evaluated. This suits particularly well lazy evaluation, as data need not be available for an expression to be activated. It is thus possible to recursively traverse a graph to extract from it data. The recursion mechanism used certainly requires the maintenance of a large stack.

The structure of this stack and its dynamic behavior are the main concerns when tackling the issue of storage management.

Conversely, the second evaluation model advances program execution by maintaining a full view of the available data. New results are produced as valid inputs are directed to computation nodes. The path data elements follow is deduced from the input code through the study of data dependencies. The dataflow model therefore clearly follows an eager evaluation approach quite appropriate to strict languages such as SISAL [45]. Languages such as Id [46] and parallel Haskell (pH) [47] are however non-strict in spite of following this model.

In both these approaches, the thread life cycles can take three forms. In the first, a parent thread blocks until it receives a *notification* from all its child threads that their work is complete. The second, fork-join model, is a particular case of the first, where all child threads are identical to one another. In the third, the threads created become completely independent of their parent. The only guarantee of synchronization in this last model is the structure of the graph.

In multiprocessor systems, the storage management model is consequent to the high latencies in communication between processors. These long delays encourage developers to try to prefer to move data as little as possible. In non-strict languages, this involves a distributed stack with several branches (a "cactus" stack) that may be quite complex to maintain. The advent of multi-core processors with very-high speed buses puts back into question such approaches.

Finally, the load distribution model can be either passieve or active. An active model has a main processor designating subordinates to the computation of certain tasks. In the passive model, a processor waits to be idle to ask for work. This may cause some extra communication overhead, as a PE asking for work needs to send a message to perform its request, and the main processor to acknowledge it before sending the context. It may, however, help make the load distribution even if a fair task dispatch policy is implemented.

2.5 Future of Parallel Programming

Consequently to the changes in hardware architecture, functional and parallel software paradigms must be re-examined. A new set of requirements must be written to lay a strong foundation for the future. Several new languages and APIs are currently emerging for several architectures and will be described and compared.

2.5.1 The Berkeley Conclusions

A large team of researchers at Berkeley [5] studied the challenges facing mutli-core architecture designers and software developers. They drew conclusions for every aspect of the problem, from the silicon process to the new programming paradigms required. Concerning the new approach required to develop software for these new systems, the following conclusions apply:

1. programmers should easily develop highly parallel applications
2. all considerations must eye a future where 1000s of cores rest on a chip
3. future programming models must abstract away the underlying architecture
4. programming models must support a wide scope of parallelism
5. microprocessor architects must sacrifice performance improvements if the use of new features is not evident to programmers
6. operating systems must undergo a dramatic re-thinking, bringing about support to virtual machines

The Berkeley team arrives at these conclusions by observing the current trend in favor of multi-core systems. As parallel microprocessors move into the mainstream, software must make use of the new resources available as it is in fact the primary goal of any hardware advancement to increase the efficiency of the code written. Developers may not be capable of taking advantage of resources available to them, either because of the difficulties of parallel programming or of a lack of knowledge of the architecture. The difficulties of present-day parallel programming thus constitute a first hurdle to overcome. Also, it occurs quite often that an intimate knowledge of the underlying architecture be required to take fully advantage of the resources available. This problem is abundantly apparent when programming for the CBE, as communication between processors is explicit and requires great knowledge of its bus. Any new approach to parallel software development must thus provide tools capable of bridging the gap between programmers' capacities and the architectures capabilities.

Further, it is paramount to keep an eye on the future of hardware. It is entirely conceivable that microprocessors available in a few generations will have thousands of cores. Such massively parallel engines would be impossible to program efficiently using current tools. Further, for these enormous chips, it is quite likely that several different core types will be present at once, each demonstrating different performance capabilities (e.g. SIMD, superscalar, VLIW, etc.). The new tools available must then be able to extract parallelism from an application at several levels: first to distribute it to clusters of cores, then amongst them, and finally at the word, byte and bit levels.

Virtual machines will become an integral step of thread management within an application, as several threads will be allocated to a "black box" by means of processor virtualization [48]. Management of such systems can become quite complex as its size affects memory coherence, I/O use and process communications, possibly requiring a significant amount of changes to modern operating systems.

2.6 Emerging Languages and Platforms

Parallel to the development of CPUs, the computational power of GPUs has dramatically increased. This development has spurred interest in using GPUs for general purpose computing, dubbed *GPGPU*. GPUs have always been used as real-time systems; a resident code handles *streaming* input to render an image. Rather than an image, it may output useful computation results. These are returned to the host CPU and Main Memory, implementing a *host/device* programming model. The remaining challenge is to dynamically send code to the GPU. The Brook programming language developed at Stanford [49], as well as NVIDA's CUDA [50], implement extensions to C as a solution. Brook is now part of AMD/ATI's solution to its future stream processors. OpenCl, a forthcoming open standard API for parallel programming with strong industrial backing, follows the same approach albeit without a new grammar.

Brook approaches the streaming model by creating a *stream* type to variables. Device *kernel* functions expect such inputs and conceptually operates upon them as they continuously arrive, in a fully parallel fashion. The inputs are typically memory arrays on which a `streamRead(source, destination)` function is applied, then fed to a kernel. For its part, CUDA requires of the programmer much knowledge of the underlying hardware. It extends C/C++ with modifiers to functions and variables to access different aspects of its hardware units. For instance, the `__host__`, `__global__` and `__device__` function modifiers respectively indicate that a function is written for the host CPU, an entry point from the CPU to the device, and an internal device function. Similarly, modifiers can specify the location of variables on the different memory hardware elements of the GPU. As in Brook, *kernels* represent device code. However, an elaborate API exists to perform inter-kernel communication. Further, kernels are launched in blocks of threads that may share memory. A limited number of such threads in a block is allowed however, and language constructs provide means to subdivide excessively large groups of threads into a conceptual 2-D grid of thread blocks.

Not to be left behind, functional languages also offer great promise. Already, Microsoft found its experimental F# language, a variation of OCaml, itself a derivative of ML, to be so promising that it will incorporate it into its .Net framework. This development is sure to be a boon to functional programming as a whole, as one of the main issues that plagued the development of functional programs is the lack of library support [51]. Unfortunately, it is still unsure what multiprogramming advancements F# will bring.

Older languages may also be revived thanks to current hardware trends. Erlang [52] (1987), a robust open source functional language developed at Ericsson has supported parallel programming for a long time using message passing. The Fortress project [53] held promise as well, but its future is rather in doubt. However, Haskell, with its large user base, holds the most promise for parallel functional languages even though multi-core implementation of its parallel versions have not yet surfaced.

2.7 Squid/SRE: an integrated three-step solution

Our approach to our multi-core programming framework design starts from the ground up. Given the intricacies and tediousness of parallel programming, our prime objective is to minimize the programmer's involvement in low-level details and to leave most of the optimization effort to the compiler itself. It follows quite logically then that the program should follow conventional single-thread programming, leaving all the partitioning, synchronization and optimization decisions to the compiler.

Our solution is certainly an extension of the functional progamming models developed for multiprocessors. It is novel however in its targeting of multicore processors, and the CBE specifically. This solution thus stands clearly opposite `gcc` that performs no data partitioning and focuses on the explotiation of ILP. Our solution offers to abstract away **all** low-level considerations in roughly the same manner as ALF. However, NCC/Squid removes the programmer's responsibility of parallelizing code. This decision to take responsibility for virtually all difficulties of CBE programming may hinder its performance, but could prove to be a good basis upon which to perform research in parallel programming. Squid studies data dependencies in much the same way as Zhao (*c.f.* Section 1.5), its analysis is enhanced through the analysis of a functional language.

As a first step, we thus propose NCC, a new programming language constructed with the aim of voiding the difficulties of multi-core programming. NCC provides constructs that simplify the compiler's code analysis while remaining intuitive to the user. The strength of NCC is its ability to expose parallelism and synchronization requirements so a compiler, Squid, may break up and optimize the programmer's code. NCC and Squid, are described in Chapter 3.

Prior exposure to Mitrion-C [54], a high-level hardware description language, provided an interesting starting point. Mitrion-C is a strict functional dataflow language that is used to generate application-specific processors on field-programmable gate arrays. We made several important modifications to the language design. Some of its particularities were kept, but its many constructs were discarded, as they either did not apply to software, or were simply tedious to use. However, NCC follows the strict dataflow functional language basis of Mitrion-C.

In a second step, we elaborate on our design of the Squid compiler. Squid intends to perform a high-level analysis of input NCC code and adapt it to multi-core architectures in general and to the Cell Broadband Engine in particular. As it stands, Squid targets the CBE specifically, although the approach it takes in studying NCC code and treating it is applicable to most multi-core architectures.

Squid parses the input code using NCC's language definition. It then analyzes the code in order to divide the input code into *tasks* and *supertasks*. Tasks are a collection of C functions that target the SPU architecture. Supertasks are C++ classes that interact with the runtime environment and direct code execution. The generation of C/C++ code allows programmers to use legacy code and libraries developed for the CBE architecture.

Given the several levels of hierarchy involved in any code, supertasks may manage Tasks, child Supertasks, and a mix of the two. The relationship between the different levels of hierarchy and within the hierarchy is very much a function of the data dependencies exposed in code analysis, and the study of these dependencies will be the basis of the runtime environment. Indeed, in order to optimize the time the SPUs stay busy, we have opted for a dynamic management of tasks and supertasks, the details of which will be elaborated on in the next chapter. This approach has several effects in compilation and particularly in task partitioning.

As a final step, we describe the Squid Runtime Environment (SRE). The SRE is responsible for runtime management of tasks and supertasks, and the interaction with the underlying hardware. The SRE follows a host/device approach. In its application to the CBE, the PPU runs two POSIX threads: the first is the host, and manages I/O and all functions that do not pertain to the accelerated code, while the second runs the SRE. The SRE is described at length in Chapter 4. The SRE acts as a portal to the accelerating SPUs. In this model, the host thread calls functions to be accelerated by the "device", by linking them to the SRE. These functions, of course, were written in NCC and are the product of Squid's compilation.

2.7.1 Why NCC/Squid/SRE

The Berkeley conclusions described in Section 2.5.1 provide a strong basis for decisions in our approach to the design of NCC:

- NCC is easy to use;
- The partitioning method used by Squid can easily extend to an increase in the number of cores;
- NCC has no construct linking it to the CBE or any other architecture;
- NCC's nested parallelism can allow it to exploit several levels of granularity;
- The two-layered approach of both language and runtime environment can adapt generated code to complex architectures;
- The SRE currently exploits existing OS resources;

These justification validate in our view the approach taken to multicore development. A complete description of the NCC language, Squid's techniques and algorithms, and the SRE's mechanisms are described in the chapters that follow.

Chapter 3

NCC/Squid

In this chapter, we shall describe the first two steps of our solution. First, high-level design decisions in the design of NCC are justified, then the syntax, operators and constructs are explained. The task partitioning mechanisms proper to Squid are then described in some detail, in order to elucidate the relationship between the division of code at compile time and its distribution at runtime.

3.1 Design of a Dataflow Language

There are typically two levels of granularity in multi-core architectures from which to exploit parallelism. The first, very coarse, distributes code and data among several processing cores. The second is usually very fine, and typically consists of taking advantage of data locality. Every program must then make intelligent use of these two granularity levels and divide the code appropriately. This work attempts to remove nearly all code partition decisions from the user. Therefore, it is very important for the compiler to benefit from the maximum amount of freedom in analyzing and partitioning the code.

If the code can be divided at virtually any point and at any depth, one must ensure that these actions do not compromise application integrity. The language must therefore respect referential transparency described in Section 2.3. Referential transparency guarantees application integrity, but is not sufficient to construct a graph that is easy to partition. Therefore, we add the requirement that the dataflow be explicit. An explicit dataflow graph built in this way represents a tree of nodes representing data.

Familiarizing with a new computer language is always a challenge, especially when the migration to functional programming imposes a paradigm shift on the developer. Learning NCC is expected to be difficult as well as it forces several constraints that are often unusual, and permits habits that are typically forbidden in most languages. For these reasons, NCC constructs will remain as close as possible to common ones such as C, MATLAB, and python.

NCC code will follow largely the same structure as most imperative languages. Functions are the highest-level constructs that may contain several statements. Functions have inputs and outputs, all of which are typed. The body of a function contains a range of statements. Every statement assigns a value to one or more variables. These values may be the result of arithmetic operations, conditional statement constructs, or iteration constructs. NCC specifies the peculiar requirement of enforcing a single-assignment rule to variables.The modalities and justifications for this important rule will be elaborated on in Section 3.2.3.

3.1.1 Variables, Data Types and Syntax

The lowest-level construct in NCC as in most imperative languages is the variable. In the frame of referentially transparent language, variables represent no more than data. The data can take the form of an n-dimensional array or may simply remain scalar. Each and every variable is associated to a type, in just the same way as it is in C. However, the variable's type declaration is not always necessary. The temporal and structural modalities of variable declarations and assignments are thoroughly discussed in Section 3.1.2.

NCC defines the same data types as ANSI C: **char, short, int, long, float, double** as these types are inherent to all modern processors. As it was previously noted, the dependency to every variable is explicit. It follows henceforth that so is its type, given a set of initial conditions. This observation voids the need to declare a variable's type in straightforward cases. Some functional languages infer types even for ambiguous cases; however, we choose to force type explicitness in all ambiguous cases.

3.1.2 Scopes

The most important level of granularity in NCC is the scope. A scope is defined as such:

Definition 1. A scope is a group of statements containing a limited number of variables, expliciting some as outputs. They can be seen as *black boxes* the output of which could be replaced by the equivalent value by virtue of referential transparency. A scope may contain other scopes as part of statements.

The scope can thus be seen as a proverbial black box with inputs and outputs, thereby enforcing referential transparency at this level. However, this definition and the associated rules are not sufficient to build a dataflow graph. Additional rules are thus necessary to be applied to statements within a scope to build such a graph *within* a given scope. It is already established that scopes can inherit variables (data) as inputs, and these will constitute entry points to the graph. Referential transparency specifies that 'states' may not exist, i.e. that a block of code may be called at different moments with different outputs must give the same outputs. Similarly, outputs must be clearly defined in every construct to provide exit points to the graph.

Since NCC must describe dataflow rather than temporal flow, there must be no ambiguity in the actual flow of the diagram. We must then consider potential sources of ambiguity. Every variable's assignment can be considered as a node in the dataflow graph. Variables are described using identifiers. These identifiers represent the variable's node throughout the scope. Therefore, performing an assignment to a given variable more than once creates several nodes identified by the same identifier. This behavior is clearly ambiguous and must be forbidden.

A more subtle ambiguity may occur in a case of circular referencing. Indeed, if two variables refer to one another in the same scope, how can the value of either be evaluated? One solution is typically to set initial values to one of the two. Assigning initial values within the scope clearly constitutes multiple assignment. Circular referencing is not strongly forbidden in NCC, and is in fact one of its most significant features. However, the danger of ambiguity forces us to strongly curtail its use.

These observations lead us to enforce the dataflow model through the following rules:

1. Any variable may be assigned to once and only once in a given scope.
2. Every construct must clearly indicate its outputs.
3. Circular referencing is not allowed with the exception of loops, in which case an initialization value must be set outside the scope.

These rules cause an important potential **relaxation** in programming use: *code flow does not need to dictate temporal execution flow.* In other words, the programmer does not need to write code in the order it is expected to be executed. Since every variable is assigned to once and represents solely data, we can represent every scope as a directed acyclic graph (DAG)

Definition 1. A directed acyclic graph, also called a DAG, is a directed graph with no directed cycles; that is, for any vertex v, there is no nonempty directed path that starts and ends on v.

Every node in the DAG represents a variable which in turn represents data, toward which a *single* edge only is directed. The temporal flow is thus implicit in the DAG, no matter the order in which it is built.

The example depicted in Figure 3.1 demonstrates how two statements written in different order render the same graph.

a = b + c;	b = h + g;
b = h + g;	a = b + c;

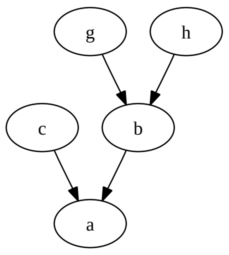

Fig. 3.1: Inverting the order of statements does not change the DAG.

Let us introduce the more complex example of matrix-vector multiplication of Figure 3.2. The code contains 3 scopes, denoted in the figure by 'scope 0-2'. Notice that the initialization to the variable `to_sum` is written *after* the statement in which it is referred. Notice also that scopes 1 and 2 are nested, creating a natural *hierarchy of scopes*. This hierarchy is important in future considerations.

```
int [100] mmul(int[100][200] matrix, int [200] vector){
   final = for(row in matrix){
        result = for(ea, eb in row, vector){
            to_sum = to_sum + ea*eb;
        }(to_sum);
        int to_sum = 0;
    }(><result);
}(final);
```

Fig. 3.2: Matrix-vector multiplication expressed in NCC.

3.2 NCC constructs

NCC intends to cover the full spectrum of functional capabilities any complete language would. For this reason, it incorporates much of the same constructs found in the usual languages, such as arrays, loops, function calls, and conditionals. In spite of being an important

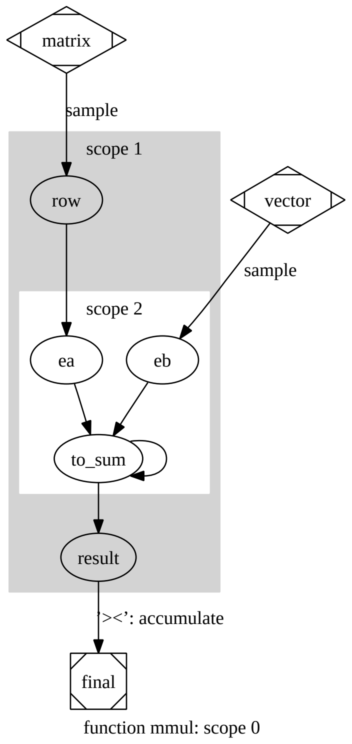

Fig. 3.3: Equivalent DAG of the code in Figure 3.2

feature in most languages, the C equivalent of the 'struct' does not exist for now in NCC, and is left as future work. Structures were left out of NCC not due to difficulties in their grammar or in the analysis of their use, but rather due to data alignment issues particular to the Cell Architecture.

3.2.1 Scalars and Arrays

NCC ascribes to every data element not only a type but a dimension, each axis of which is described by a range, much in the same way as Java. It was already noted however that NCC does not permit pointer arithmetic and, by extension, pointer referencing. Arrays are thus handled by the programmer as data elements in the exact same way. Several operators are necessary to work with arrays to perform such operations as concatenation, subspanning, and multi-dimensional construction.

Concatenation

In the first case, it is quite common for scalars, arrays, or a mix thereof to be merged into a single array of elements to be processed further. Thereby arises the need for a concatenation operator, in this case Mitrion-C's '><':

```
a = c >< b;
```

where `b` and `c` are arrays of the same dimension. If `b` and `c` are multidimensional, every dimension's range must be the same so as to create an n-dimensional cubic matrix.

Arrays may also be merged by keeping their 'identity'. To that end, they can be included in a new array of arrays, that is, a multi-dimensional array, where each dimension represents a collection of arrays, each of which capable of being a collection itself. To build multi-dimensional arrays, NCC uses a bracket notation:

```
variable2D = [ v1D1, v1D2 ];
```

Range and Subspanning

Finally, it is of little use to build up vectors if parts of them may not be accessed. Similarly to MATLAB, NCC allows access to array parts through individual access and subspanning.

To access an individual element of a variable's array, the bracket notation is used after the variable's identifier:

```
a = multidim[2][4][i];
```

As in all languages, the first bracket group represents the most significant dimension.

NCC further describes a simple way to automatically generate an array spanning a range:

```
[ X1 .. X2 ]
```

where X1 may or may not be larger than X2, but must both be constants. This span can be used as data elements as well as array indexing. Its use as data elements is particularly important using loops (*c.f.* Section 3.2.3).

3.2.2 Basic Expressions

In the simplest case, variables are assigned the result of arithmetic operations. The arithmetic operations and operators defined in NCC are exactly the same as those of ANSI-C, so as not to confuse a new programmer. Arithmetic operations are very simple to analyze from a dataflow perspective, as all variables referenced are inputs to the variable (node) being assigned.

3.2.3 Loops

NCC defines two kinds of loops: the 'for' loop, and the 'while' loop. They differ slightly from other languages in their form, but also in their use. Typically, loops can be used in two ways:

- The number of loop iterations is known at compile time
- The loop count cannot be determined at compile-time

In the first case, several techniques have been developed to optimize performance, such as loop unrolling, efficient vector processing, etc. Further, it is quite often the case that the values of each iteration is saved, building up a vector of loop-assigned data. In that case, the loop size corresponds exactly to that of the vector. Therefore, contrary to languages such as C, the 'for' loop iteration size must be known at compile time to empower the compiler to easily apply optimization techniques.

In the second case, most if not all of these optimization techniques are no longer applicable. Further, an array cannot be efficiently re-sized across loop iterations for it to accumulate values. Therefore, NCC requires of the programmer to convert to 'while' loops in the common way these 'for' loops written in languages such as C/C++.

For example, the C/C++/Java loop:

```
for(i=0; i < b; i++){
    ...
    out = ...
}
```

can be converted to:

```
int i = 0;
out = while(i<b){
    ...
    i = i + 1;
    ...
    a = ...;
}(a);
```

A typical use of loops involves using the result of one iteration to compute that of the next. The requirement forbidding any circular dependency in addition to that of single-assignment thus pose a problem in this case. Therefore, in the special case of loops, a minor digression is allowed; a variable may be written to several times insomuch as it is across several iterations. However, the value read in that scope will be that of the previous iteration rather than that just computed. Therefore, the acute observer will understand that there is no 'circle' in the dependency graph, but rather a spiral across the iteration axis.

However, this solution still causes an initial condition problem. What is returned by the first read? The only way to solve this question is to require every *loop-carried dependency* (LCD) to be initialized in the scope exterior to that of the loop. For much the same reason, it must also necessarily be typed.

Let us re-examine the matrix-vector multiplication example of Figure 3.2. The variable `to_sum` is a LCD across the innermost loop. Unrolling the loop for a few iterations offers the graph of Figure 3.4.

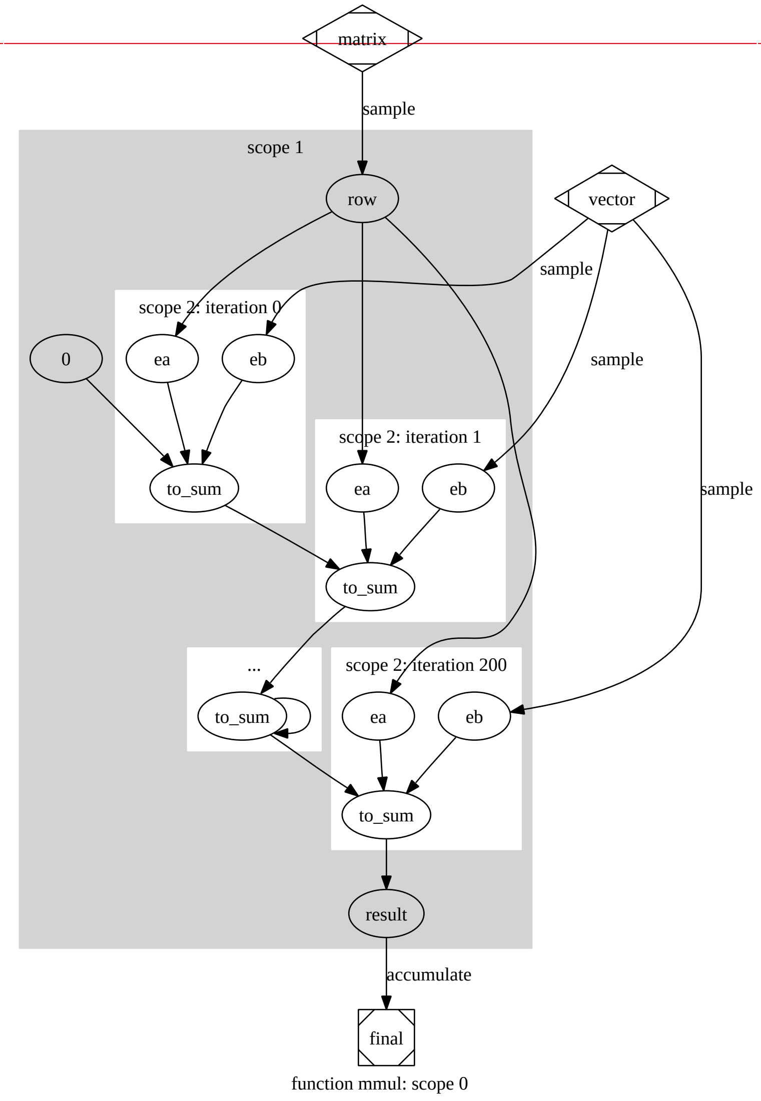

Fig. 3.4: Matrix vector multiplication example with unrolled inner loop.

These solutions bring about an important feature of NCC and a clear mark away from functional languages. Indeed, the core property of functional languages is to be free of states. However, LCD provide a sure way for a program to symbolically change state across some loop's iterations.

The syntax of loops is as follows. For the simple while loops:

```
a,b,... = while(condition [by loop_index]){
    x = ...;
    some code
    y = ...;
}(y, ...);
```

A ‘for’ loop’s syntax, however, adds an element of freedom. A loop with a definite width can overwrite its result or store the individual result of each iteration in an array. NCC allows such behavior by offering two operators. The first, ‘><’ modifier, copied from Mitrion-C, can precede a variable in an output clause to indicate that the output variable’s value across all iterations are accumulated in an array. The receiving variable thus has one more dimension than the output variable, with a width equal to the number of loop iterations. For instance, if output variables are two-dimensional, the receiving variable will be 3-dimensional.

The second collection operator is not sourced from Mitrion-C, and arguably fills a gap in the language. The ‘<>’ modifier, concatenates output results in a single array; if individual output variables have 2 dimensions, so will the receiving variable. In the absence of either ‘<>’ or ‘><’, the *last* result of the output variable is returned. All three cases are permitted both for parallel computations and loops containing LCD.

3.2.4 Functions

In the same way as ANSI-C, functions constitute the highest-level construct in NCC, as no classes are currently defined. Functions preform essentially the same purpose in NCC as in virtually any other language. They provide the user the ability to call into execution a block of code. NCC follows the example of MATLAB and other languages in requiring clearly the number of inputs and outputs. Inputs as well as outputs must be typed. Typed inputs provide the root information required to disambiguate types for all the references made to them. Outputs are required to be typed as a precaution for the user.

A function declaration follows the following template:

```
type [[size]] [type [[size]] ...] function_name( type [[size]] argument, ...){
```

Functions operate in a way similar to python and MATLAB, as they may return more than one variable. In a function call, however, NCC does not yet allow for a variable to receive 'tuples', that is more than one variable symbolically grouped into one. In the case where more than one variable is returned, more than one receiving variable must be specified, taking into account the possible type conflict. Further, functions returning more than one variable cannot be referred to in arithmetic operations because of the understandable ambiguity.

```
 v1,v2, ... = function name( inputs );
```

3.3 Block Partitioning

NCC rules such as explicit dependencies and single assignment not only enforce a rigid structure horizontally between blocks of code, but also vertically, within the hierarchy of scopes. From function scope to inner scopes, dependencies are somewhat rigid and well-determined at compile-time. This hierarchy structure can be seen as a natural boundary and a useful one when partitioning the code between supertask and task, as well as between supertasks. Therefore, Squid can easily determine a scope (and its substructures) to be a good candidate to be rigidly fixed into an SPU task by evaluating different properties inherent to that scope.

Several factors can influence the decision that a given scope represents a task: the amount of code, the amount of stack space required, the amount of input and output data, the amount of time spent in the code are all important factors. The various properties of the target architecture thus become the prime source of block partitioning decision-making.

3.3.1 Decisions

In order to decide intelligently on the task partitioning criteria, it is necessary to study the system's constraints. Since a task is executed on an SPE, the SPE's restrictions must guide the partitioning decisions. As mentioned in the Background (Section 1.2.2, the SPE is very powerful for vector computations, but poor for scalar and conditional ones. Further, its 256

KB of memory represent a small space in which to hold code, heap, and stack. This space is further reduced when techniques such as multiple-buffering are considered (*c.f* Section 1.4). For this reason, we hold the memory restriction as the primary restriction in all block partitioning considerations.

However, execution time usually holds much sway in task partitioning in multiprocessor systems. Indeed, synchronization points usually imply that time is wasted waiting on a task. Why are we taking such a different approach? Squid operates on the underlying assumption that the input code is very large and will run for a significant amount of time. Further, such a large code will in all likelyhood have several tasks simultaneously ready for dispatch. Given the SPE's computing power, the more code is executed therein, the faster the execution, especially given the SPE's vector processing capabilities. Therefore, in our view, the SPE's computing power reduces significantly the importance of the time spent on the task on overall performance.

This conclusion stems from the analysis of three possible scenarios. A task may spend most of its time in:

1. A large and completely parallel 'for' loop.
2. LCD's across the numerous iterations of a 'for' loop.
3. A 'while' loop of unknown size.

The third case is the simplest one to discard. Indeed, a 'while' loop intrinsically cannot be divided nor can its computation time evaluated in advance. In the second case, it is quite evident that it is best for the execution to remain on the SPE because the loop can only be evaluated serially. Every attempt to 'switch out' of the device is bound to cause a needless I/O penalty.

The first case commands however more attention. If a loop is fully parallel and can be expected to run for a long time, why not cut it in very small pieces to bring down that time? The answer is twofold: first, a very parallel code is already accelerated by the vector processing of the SPE's. Second, the overhead caused by setup time on the SPE, however small it may be, accumulates more rapidly if the number of tasks increases. Hence, making the largest possible tasks is a good rule of thumb to optimize performance. Naturally, more advanced techniques have been studied in the literature and could be applied in this case. These should be actively pursued in any future work.

3.3.2 Multilevel Graph Partitioning

If SPE tasks constitute the basic building blocks of computation, one could choose to move to a 'flat' structure consisting of all possible tasks, anticipating the need of others for itself upon completion. Beyond the fact that such an approach would be very difficult to implement, it would disregard all the hierarchy intrinsically built by the user while writing code. This superstructure contains quite a large amount of circumstantial information, such as loop structures, horizontal and vertical dependencies, etc. This information is largely hierarchical and provides a great basis on which to organize data flow. Scopes are always associated to a behavior, as they may encapsulate the body of a function, loop or conditional statement possibility. In all these cases, it is possible to create supertasks associated to that behavior.

Squid thus performs several iterations over the code structure generated by the initial parsing. It identifies all dependencies in the code, from scope to scope vertically and within scopes horizontally. Based on the information thus gleaned, it recursively looks down from scope to subscope attempting to fix a scope into a task. The criteria applied is an estimate of the weight of the input and output data and is termed the *memory criterion.* The top-down search performed attempts to fix into a task a scope of the highest possible order in the hierarchy, so long as it respects the memory criterion. This approach is a simple way of maximizing the amount of code held by a scope and grossly minimizing the amount of total I/O. Indeed, the higher up the hierarchy the scope, the more subscopes does it contain, typically in the form of an inverted tree.

When parsing the tree to fix a task, Squid may encounter situations where the scope body is quite insignificant but might iterate over a very large amount of data. Indeed, in typical applications, 80% of the computation time is spent in small loop kernels. In the example:

```
int [1000000] loopalot( int [1000000] a, int [1000000] b){
    out = for(c, d in a,b){
        o = c + d;
    } ><o;
}out;
```

the loop iterates a million times. This loop is certainly too large to fit in a single SPU, as the amount of input data is 2*4B*1 million = 8MB, in addition to another 4MB output. However, the inner loop code only requires 12B. Launching this task a million times will incur a disproportional time loss in overhead. Therefore, in such a case, the loop should be broken down in several maximum-fit parts. This is the principal reason for NCC's strictness on hard loop bounds. Since loops' breadth are known at compile-time, loop partitioning is relatively straightforward.

This approach still leaves some ambiguity in the case of the 'for' loop with LCD's. On the one hand, its I/O signature is very similar to the parallel case. On the other, no iteration block can be started before the precedent is completed. One could propose a 'break' in the task block model: keeping the task running and swap I/O to mask task management overhead. As it stands, Squid maintains the task-block model, in the hope that the time lost can be masked by the execution of different independent tasks.

Let us re-examine our matrix-vector multiplication example. There are several ways imaginable to partition this loop, depending on the criteria established. Here is a list of possibilities in the order of further partitioning:

- A single task containing the entire function
- The outermost for loop computing a block of several rows in a single task

- Each row being computed as a task
- Parts of a row are computed in a task, with the result carried over as an LCD

This list disregards the case of a single multiplication per task as too inefficient to be considered a task. The last case is certainly the most complex, and is described by Figure 3.5.

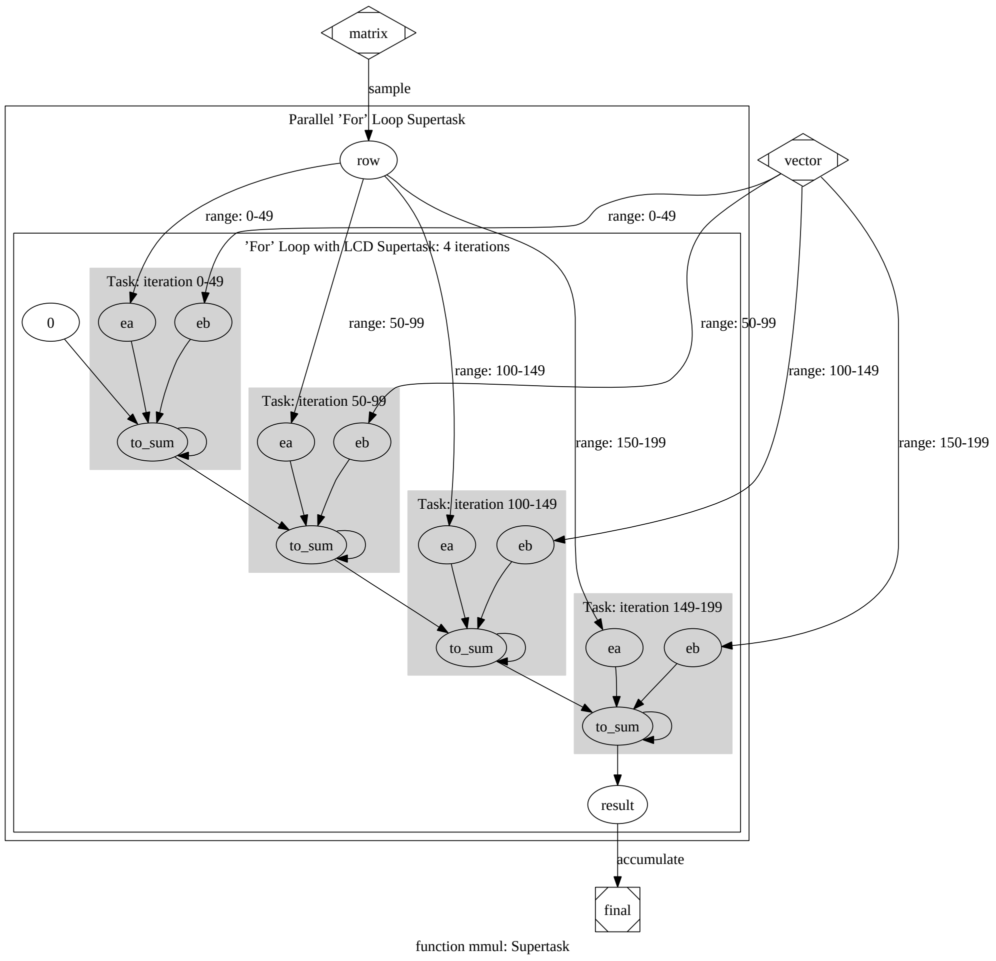

Fig. 3.5: Matrix vector multiplication example with partitioned inner loop.

A possibly fair compromise solution probably lays in a further loop partitioning. Not only would the loop be split 'vertically' in the hierarchy of scopes, but it could be split 'horizontally' as well. In most cases of loops with LCD's, a certain part of the loop is parallel among all iterations. Merging these fully parallel parts can offer in-time feeding of the LCD task, perhaps even hiding much of the writeback time lost through the overlap of the parallel task and LCD one. This approach is currently not implemented, and remains a priority in any future work.

The analysis leading to the fixing of code into a task only takes into account the amount of data *necessary* for that task. It does not however account for the mechanisms and subtleties of dividing data and distributing it. This division of data will be shown to cause extra I/O than the minimum evaluated. The SPE memory structure described in the next chapter specifies a fixed maximum-memory buffer for every task, over which there may not be any overflow. Therefore, the task partitioning memory criterion should force a lower bound on the available memory to account for the possibility that the *actual* I/O caused by data partitioning does not overflow the SPU buffer. Once block sizes are fixed, they should be checked for any overflow. Conversely, a minimum task size threshold criteria is also used to determine whether a code block is too small to be set as a task.

3.3.3 Array Quantization

Task and supertask objects are meant to be created and managed dynamically, with the consequence that data itself is allocated dynamically. Instead of creating huge sparsely filled arrays, Squid allocates data blocks in main memory as tasks are created. This fragmentation of data into **quantas** has an important consequence: array referencing must thereafter take into account block boundaries. Taking for example an array partitioned in blocks of 100 elements, a reference spanning 200 elements with a non-zero offset will require not two but three blocks. Squid must thus account for this possibility when studying the task partitioning.

Let us study the example of the simple FIR filter described by the code below:

```
float [200000] FIR(float [200010] input, float[10] K ){
    result = for (i in [10 .. 200009]){
        product = for (j in [0 .. 9]){
            element = K[j] * input[(i-j-1)];
        }(><element);
        float sum = 0.0;
        osum = for (r in product){
            sum = sum + r;
        }(sum);
    }(><osum);
}(result);
```

This case is not simple to partition because of the overlapping dependency on the `input` variable. Indeed, for each iteration in `i`, this dependency overlaps 9 items with the previous. Cutting the `input` variable in two will cause both blocks to be required when values of `i` attain the cutting point. Quantizing the `input` array must therefore take into account the effect of this overlap.

Hence, in order to quantize array blocks, once a scope or part thereof is fixed as a task, Squid must study the scope tree in two passes. In the first, it scans the scope tree by parsing from the bottom up to evaluate how best to cut individual arrays. Indeed, data may be created in large blocks but consumed in smaller ones or vice versa, as our example demonstrates. A careful balance must be made to ensure that array blocks referenced are not disproportionally large to consumers, and that data blocks created do not cause the task where they are created to be deliberately small. The array blocks thus created are quantized for all intents and purposes throughout the code. A final pass down the tree fixes the agreed upon size to all references to those elements.

To cut up tasks' array references in blocks, the affinity test described in Section 2.1 is performed. The dependency size thus depends on the range of the index variables over which loop code references the array. This analysis applies to all loop indexes placed *below* the cut location. The cut size is thus the sum of the span of all such loop index variables in the array. In the case where the reference's array index is not affine with respect to loop variables, the breadth of the array referenced cannot be known. In this case, the whole array is required as an input.

Referring back to the FIR filter code, it can be observed that it can be cut at the level of the loop indexed by `i` in 40 parts, spanning each 5 thousand iterations. The array reference thus spans 5,000 iterations in i, plus 10 in j, totalling 5,009 elements (1 is subtracted because of overlap in the initial value of `j`).

This analysis is however not sufficient to determine the code block required by a given dependency, as it merely indicates its size. To determine the location of the block in question, the offset must be taken into account as well. Two possibilities may arise when studying this question. In the simpler case, the offset is merely a constant number. However, it may also be a variable. In this latter case, it is quite evident that the variable in question must be inherited as an input to the task, as there would otherwise be no way to identify the block required before the task computation has started. We must then enforce that the reference to any variable, offset or not, below the cut location would cause the reference to not be affine. This input variable will be evaluated by the runtime environment for the purpose of feeding the task the appropriate block.

Once the offset is identified, the array reference must be analyzed in light of the quantization already performed on the array. If the offset is a constant, it might or not be aligned to the block size. In the case it is not, one more block than the number expected could be required because of the lack of alignment. If the offset is a variable, the extra block is always needed, as the chance of it being aligned is expected to be random. In the FIR filter case, clearly a further block is needed, and the dependency size doubles to 10,018 elements.

Whether the offset contains an input variable or not, several references to the array may be made with different fixed offsets. Given the high likelyhood of proximity and overlap between the blocks so indexed, Squid groups adjacent and overlapping references into a single block, for each offset variable found or not. A common-sense check is later performed to ensure that the amount of referencing and consequent block creation does not exceed in size the array's own dimension.

In the final pass, the tasks are re-evaluated in light of all the information thus uncovered. The tasks no longer reference a long array but quantas of data. The references must then be modified to be adapted to reference the appropriate quanta, with of course the re-evaluated offset. The task built in the FIR filter example will thus become the following:

```
result = for (i in [0 .. 5000]){
    product = for (j in [0 .. 9]){
        element = K[j] * input[((i-j-1)+5000)/5000][(i-j-1)%5000];
    }(><element);
    float sum = 0.0;
    osum = for (r in product){
        sum = sum + r;
    }(sum);
}(><osum);
```

Where the `input` variable became a two-dimensional vector, the first dimension of which indexes the block number of he array.

3.4 Future Work

The development of NCC and Squid has borne results but is nonetheless far from over, as several parts of our framework are left as future work. NCC grammar should be extended to include such functionality as structures, and one may contemplate the creation of objects and classes. However, it would be particularly interesting to implement function pointers, offering some flexibility to the programmer.

Squid, in turn, should be extended to complete the tasks most compilers do. After identifying and studying tasks and supertasks, Squid prints those in C and C++ respectively, without performing any optimization, leaving this work to gcc. The case of SPU tasks specifically is of particular importance, as most of the information gathered in our code analysis is easily applicable to low-level optimization. Further, Squid makes no attempt at using the SIMD capabilities of the SPUs.

Chapter 4

The Squid Runtime Environment

Chapter 3 describes how Squid partitions NCC code in tasks and supertasks from its analysis of the code. They are constructed in hierarchical blocks in which a function is necessarily a supertask of the highest order and includes any mix of tasks and supertasks. Supertasks contain supertasks as well as tasks. Tasks are computational blocks of the lowest order.

The runtime behavior of the program, however, is only implicitly expressed by this hierarchy and does not express the temporal behavior of tasks and supertasks: in what order will tasks be executed? Which part of a supertask fork will have priority? It also does not express the spatial behavior of the code: on what processor will a task be launched? Where will the data be stored?

The standard answer to these questions is a logical extension of the imperative model of programming described in Section 2.3. In this model, computation is also divided into tasks. However, the dispatch and management of these tasks is fixed at compile-time, and is ordered *temporally*. The rigidity of this model requires a very large amount of synchronization considerations that are usually forced unto the programmer. It also encourages the latter to opt for a breadth-first approach to computation, where a large amount of time elapses between the creation and consumption of loop data blocks.

The dataflow model developed by the programmer with NCC, and its analysis by Squid, open the possibility of a much more flexible approach. Instead of fixing the hierarchy as a means of guaranteeing correctness, the dataflow model allows several parts of the DAG to be active **simultaneously**. We thus opt for a **dynamic** distribution model where tasks are dispatched not according to any order determined at compile-time, but solely on the availability of that task's inputs.

Object-oriented programming is quite a suitable means to describe the dataflow model. Indeed, tasks and supertasks can be seen as dataflow *objects*, where data blocks are member variables. These objects are created dynamically as the execution progresses. The speed and modularity of C/C++ is a particularly good platform in this respect.

Supertasks henceforth will not *direct* program flow, but rather **describe** data flow. The Squid Runtime Environment (SRE) will take charge of dictating the interaction between all the constructs in the program, managing memory use in both Main Memory and SPU local store, and direct the execution of tasks on the available cores.

The SRE is in fact composed of two very distinct parts that work in symbiosis. The first runs on the PPU, and is responsible for task dispatch on one hand, and task and supertask management on the other. The second runs on the SPU, and organizes the fetching and writeback of data as well as the task execution. The SRE will henceforth be referred to as the PPU SRE and the SPU SRE to avoid confusion.

In order to clearly describe the SRE, we shall do so in three parts. In the first, we describe an example problem, the Pease Fast Fourier Transform. We choose this specific problem as the structure of its dataflow sheds light on the dynamic nature of our task distribution approach. It will thus be used throughout this chapter to clarify our description of the SRE. In the second part of this chapter, the SRE's anatomy will be exposed both for the PPU and SPU. Finally, the mechanics of the dynamic management of the execution will be explained.

4.1 Pease FFT in NCC/Squid

The Fast Fourier Transform (FFT) is a very important algorithm in signal processing and poses interesting difficulties in the frame of parallel computing. It transforms signals into their frequency content, in a 1:1 mapping operation. FFT is an optimization of the discrete fourier transform:

$$X_i = \sum_{k}^{N-1} x_k e^{-2\pi jik/N} \tag{4.1}$$

Applying Equation 4.1 results in a computation of order $O(N^2)$. Cooley and Tukey [55] designed an algorithm that takes advantage of redundant computations to accelerate the problem to an order of $Nlog(N)$. Interesting work has adapted the FFT computation to the CBE architecture, with good results [56, 57].

The popular Cooley-Tukey algorithm has an irregular data access pattern that does not suit it well the task partitionning of Squid. However, a variant developped by Pease [58] has a regular data access pattern with stride one access to inputs. We shall attempt to use the example of this algorithm to help explain the different aspects of the SRE.

Let us first describe the 128k point Pease FFT algorithm using its equivalent code in NCC written below. The dataflow of an 8-point Pease FFT is depicted in Figure 4.1:

```
#define SIZE 128*1024
#define LOG_DATA 17

float [SIZE][2] pease(float [SIZE][2] x){
    // calculating the base twiddle factor coefficients
    W = for(i in [LOG_DATA-2 .. 0]){
        w = complex_exponential(2*PI*(1<<i)/SIZE);
    }(><w);

    // initializing the loop-carried dependency
    float Z[SIZE][2] = x;

    // outer loop:
    Z = for(i in [0 ..LOG_DATA-1]){
        // inner loop:
```

```
        Z_interim = for(j in [0 .. SIZE/2 - 1]){
            int index = 0;
            float [2] w = [1.0, 1.0];
            // to calculate the twiddle factor for this butterfly
            w = while(index < i){
                // compute w as a function of W using bit analysis
                t = ...;
            }t;

            // butterfly calculations:
            z00 = Z[j][0] + Z[j+SIZE/2][0];
            z01 = Z[j][1] + Z[j+SIZE/2][1];
            z0 = z00 >< z01;
            z10 = w[0]*(Z[j][0] - Z[j+SIZE/2][0]) - w[1]*(Z[j][1] - Z[j+SIZE/2][1]);
            z10 = w[1]*(Z[j][0] - Z[j+SIZE/2][0]) + w[0]*(Z[j][1] - Z[j+SIZE/2][1]);
            z1 = z10 >< z11;
            z = z0 >< z1;
        }(<> z);
    }Z_interim;

    // final bit index permutation
    out = for(k in [0 .. SIZE-1]){
        o = ...
    }(>< o);
}out;
```

It can easily be observed that the code just listed can be divided in three main parts: the first computes a few twiddle factors, the second performs the iterative butterfly operations, and the third the required data reordering.

One could also notice that the method of computing the twiddle factors appropriate to each butterfly is unusual. Typically, the whole range of twiddle factors is computed before the butterfly operations commence. The appropriate twiddle factor in the list is then loaded in time to perform the butterfly computation. In this application however, the breadth of twiddle factors becomes a limiting factor. The index of the twiddle factor is not at all affine. There is thus no way for Squid to fetch the appropriate chunk of elements from Main Memory onto SPU Local Store. In this case, the entire array would be fetched. In the case of a 128 thousand point computation, the size of the twiddle factor array is prohibitively large. We

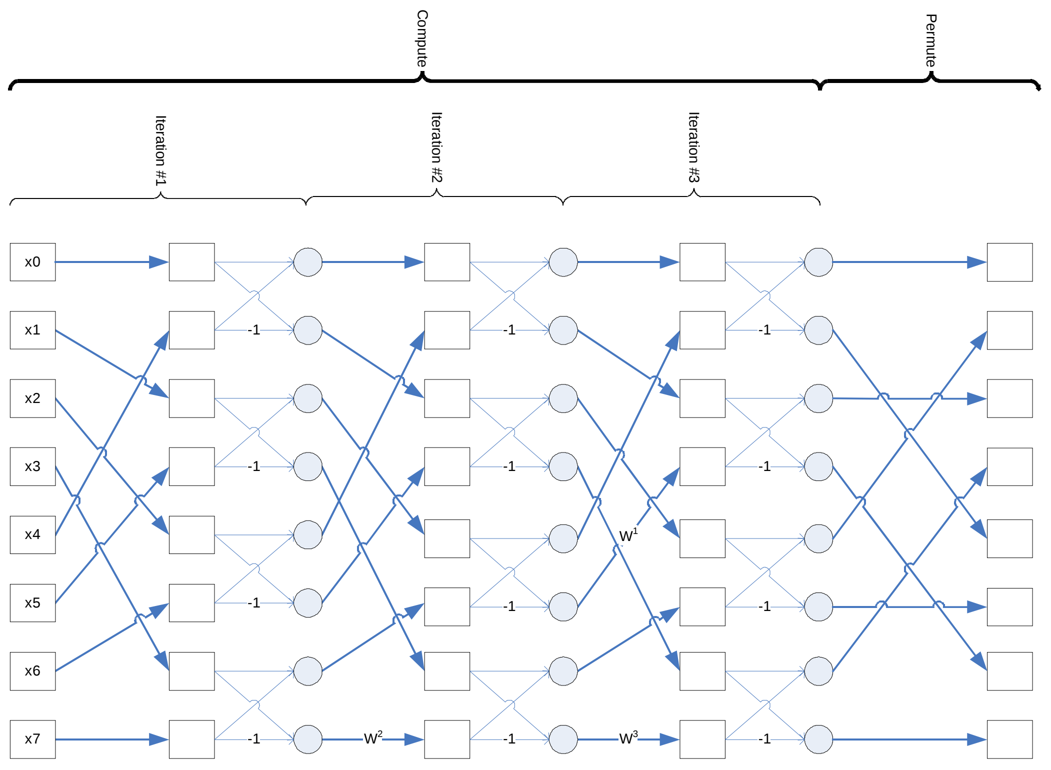

Fig. 4.1: Dataflow of an 8-point Pease FFT

thus settled for a compromise solution; a minimal set of factors are computed and stored in the `W` array. The factors needed for the twiddle factors are derived from the set as any factor is a linear combination thereof.

Three top-level scopes can easily be identified as related to the variables `W, Z` and `out`. The most interesting of those is `Z`, as it represents the main body of the computation. One can clearly observe that `W` is quite small and may not warrant the use of an SPU as there are only $log_2(128 * 1024) = 17$ elements to compute. The `out` variable may not be computed on an SPU either, as it clearly refers to `Z` with an index that is not affine. The scope that will require the most analysis will thus be that of `Z`.

We set this example such that Squid will only build low-level tasks from the scope of variable `Z`. This scope contains a subordinate scope that spans quite a large number of iterations. It requires a great amount of inputs. Beyond the base twiddle factors, it requires for each

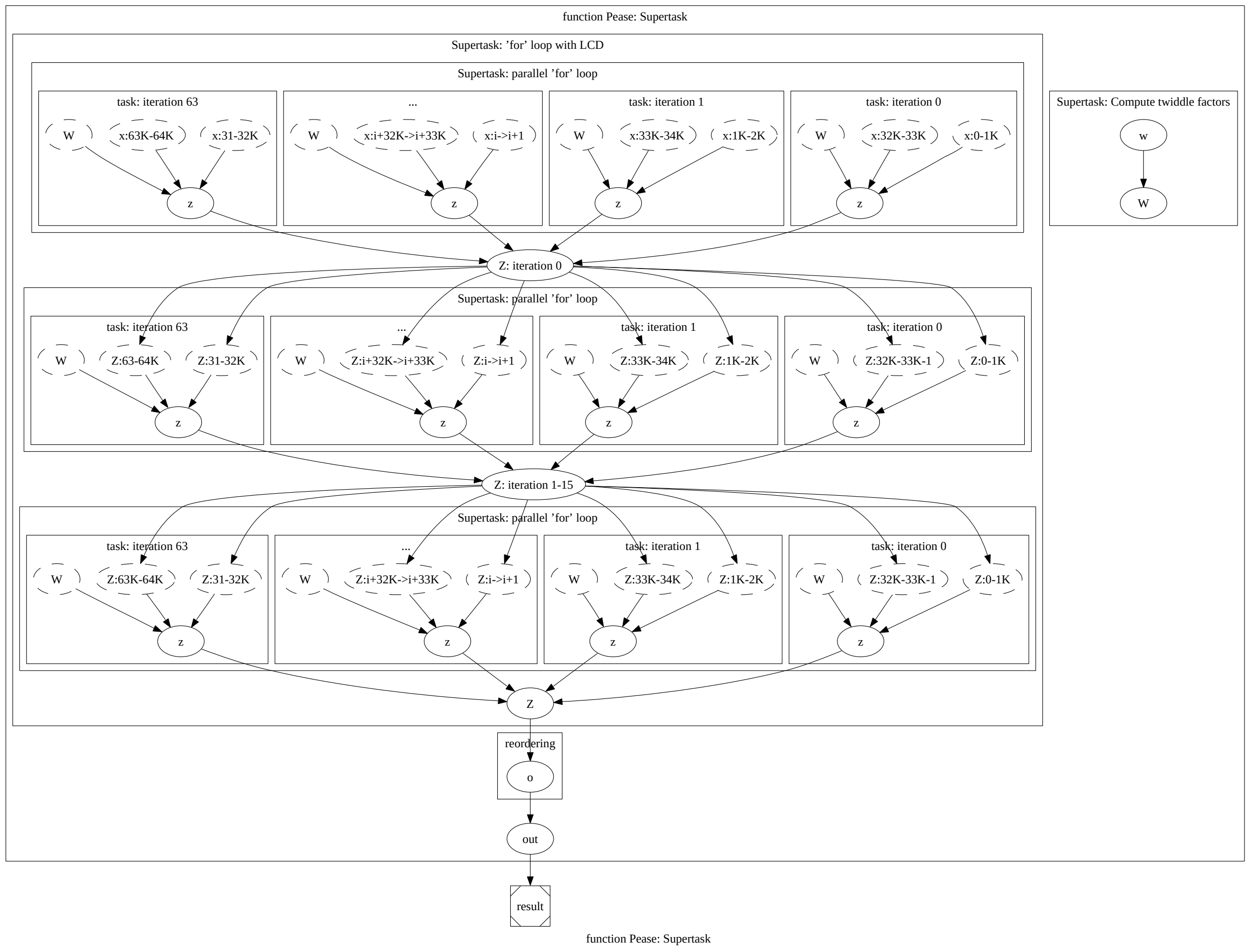

Fig. 4.2: The Pease FFT algorithm divided by Squid

iteration two elements of the previous iteration of the base array. There is clearly an opportunity here for loop partitioning. For the sake of this analysis, it was decided that the loop will be divided in 64 parts. Each part will thus require 1024 elements from each dependency, totalling 2*1024 elements. Since each element is a complex number composed of two `float` types, the sum grows to 16KB. Each block of iterations generates as many elements as it consumes, to which the 17 elements of W are added. The total rounds to a bit more than 32KB and will fit quite well an SPU task buffer (*c.f.* 4.3). The code generated by Squid will create the DAG of Figure 4.2.

4.2 Anatomy of the SRE

We remain quite mindful of the limitations of NCC in practice, especially the absence of I/O functionality, as well as its emphasis on non-temporal execution. To palliate to these shortcomings, our approach to multi-core acceleration follows a host/device model. This model will allow programmers to perform timing-critical tasks in parallel to the execution of the function being accelerated. We believe this approach will offer users the utmost flexibility in application development. Therefore, the SRE will represent the device that receives NCC functions, and a main thread of C++ code is reserved for the user to manage I/O and other functionality. The case of the Pease FFT is shown in Figure 4.3.

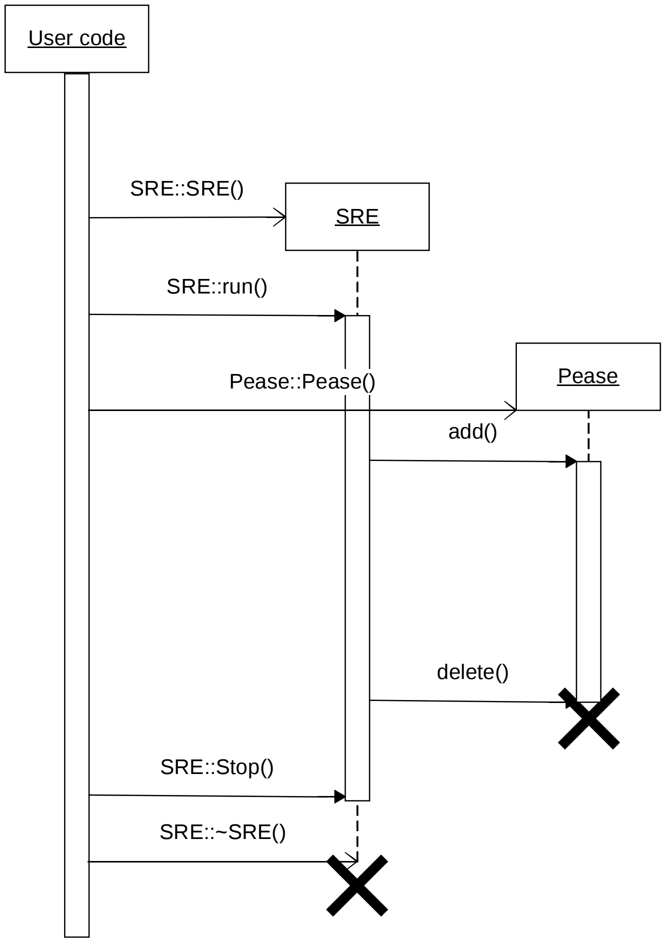

Fig. 4.3: UML sequence diagram of the execution of the Pease() function under the SRE.

The PPU's architecture is convenient for this device/host model as it is designed to work on two threads simultaneously. The SRE will thus reside on one of these threads while the user program continues on the other; once the SRE is instantiated, its execution is launched as a new thread that remains active until the user program specifies it is no longer needed. To avoid a race condition between such a command and SRE execution, the SRE will ensure all functions passed to it are completed. An emergency SRE shutdown function may also be created in the future to abruptly terminate all execution, but is currently not implemented.

Before looking any more deeply at the structure of the SRE on the PPU, let us first examine its structure on the SPU, as it is after all the purpose of this work to optimize the use of the SPUs.

4.3 SPU anatomy

The primary element of concern when dealing with SPUs is memory. Indeed, it was described in Chapter 3 how memory was the primary concern in partitioning decisions. The multiple buffering technique described below adds to these concerns, as there are now several tasks' worth in memory simultaneously dedicated in one SPU's local store. The memory management mechanism of the SPU SRE must then be carefully planned, as explained below.

4.3.1 Multiple Buffering

Traditionally, multiple buffering is used on a per-variable basis, where the next data block for an array is fetched while one is being consumed. In this framework, multiple buffering is not used to access several successive parts of an array, but is used at a higher level of granularity, the task itself. Hence, one task's inputs are being downloaded while an active task, the input of which have already been downloaded, is being executed. Data writeback overlaps the execution of the other task as well. This approach allows Squid to print C code for a task independently of the underlying architecture, extending the range of our framework to other architectures in the future. More importantly though, it is fully compatible with a dynamic model of execution where the task being computed can be seen as a black box managed by its parent supertask.

The SPU SRE thus reserves a fixed amount of memory that will be allocated to tasks during runtime. This memory will be divided in a fixed number of equal parts, thus allowing several tasks to be present on the SPU at once in order to perform multiple-buffering. In our example, each SPU contains 4 buffers. The SPU's buffers may take the form depicted in Figure 4.4.

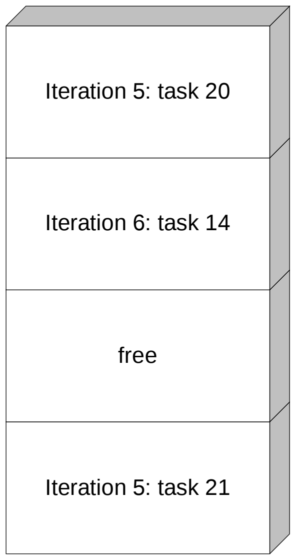

Fig. 4.4: An example state of 4 buffers running the Pease FFT example.

The number of buffers used must be fixed at compile-time. In fact, it needs only be fixed when instantiating the SRE. However, the number of buffers has a direct bearing on the size of these buffers given the fixed size of SPE local store. Further, the size of these buffers has a direct impact on task partitioning by Squid. In our example, 64*1024 iterations are performed in the inner loop and 32B of affine dependency are necessary for each iteration. In this implementation, we warrant the use of 200KB for the buffers, as our application requires a very small amount of stack. If 2 buffers are used, it warrants 100KB per task, allowing roughly 3 thousand iterations. However, if 4 buffers are used, only about 1,500 iterations can form a task. It necessarily follows that the size of the buffers must be fixed by the user at compile-time, typically offered as a parameter to Squid. In order to use simple numbers, this implementation of the Pease FFT iterates 1024 times for every task as described previously, even if it fails to fill the buffers.

Quite surprisingly, the location of the buffers had to be decided carefully. They may be situated in two places: on the stack or in the heap. Placing them in the heap may cause a dangerous situation. Indeed, contrary to most advanced processors, the SPUs do not offer any sort of protection against heap and stack overflow. An attempt to assign the large buffers required in the heap has led to the unusual situation where the stack's pointers were being overwritten as data was processed. The safest solution is thus to place the buffers on the stack, safely guarding the system from dangerous unusual behavior.

4.3.2 Functions

A function may include several types of tasks, each of which could be called several times with varying inputs. In order to optimize the use of all resources during program execution, every type of task must run on the first available SPU buffer. As a consequence of doing so, every task's code must reside on every SPU permanently. In the case where a large number of tasks are designed, the amount of memory associated to these tasks becomes a concern. Indeed, large tasks can reach a code size of tens of kilobytes, hardly anything to gloss over. The SPU SRE itself is about 8KB in size. In this circumstance, code overlay described in 1.2.2 becomes necessary. The SPU SRE would download a task's code along with its inputs. As it stands, there is no implementation of overlay in any form by Squid. It is however a very important element of design, and is a top priority in future work.

4.3.3 The Buffer

An SPU has no advance knowledge of the task it will be assigned nor can it predict the source of its inputs and other information. For this reason, every time a task is assigned, the SPU must receive with it all the circumstantial information pertaining to the task. First, the buffer should be associated to one of the possible tasks' code. Second, it must be able to singal completion to the SRE on the PPU. Finally, it must know the addresses from which to download input data and to which to upload the outputs. This information is gathered into the **task buffer** that is formed by the PPU SRE and fetched at the appropriate time (*c.f.* 4.5.1) by the SPU SRE.

The input data is fetched once, has a predetermined structure and all its data is guaranteed to be available by the PPU SRE(*c.f.* 4.4.6). Instead of setting up several memory transfers, the simplest method of fetching input data is by DMA list (*c.f.* 1.3). The PPU SRE thus fills the elements of a DMA list that contains the addresses of all input data blocks, the structure of which is fixed at compile time. The SPU SRE may thence download all the data required for a task at once. All the data will then be fetched by the SPU SRE in one command. The same reasoning applies as well to outputs; the PPU SRE forms an output DMA list that is used by the SPU SRE to send back the task's output data to locations in memory that were already allocated by the PPU SRE.

The Pease FFT tasks receive two inputs, one corresponding to each dependency on `Z`. It generates two outputs, each of which becomes a value of `Z` for the next iteration. The structures below describe the DMA lists used for the butterfly task of our example:

```
typedef struct{
    int input_top_size;
    int input_top;
    int input_bot_size;
    int input_bot;
    int W_size;
    int W;
}fft1k_dmalist_in;

typedef struct{
    int output1_size;
    int output1;
    int output2_size;
    int output2;
}fft1k_dmalist_out;
```

4.3.4 Multiple Buffering Control

At a higher level, the SPU must be able to coordinate with the SRE the management of tasks. Therefore, it needs a bit more information than merely that specific to each buffer. First, to load a task buffer, it must know where to fetch it. It must also know whether to fetch it, i.e. whether a buffer is available for computation. Finally, it should be able to signal the PPU SRE it may assign a new task to this buffer. This information is described by the following C structure:

```
typedef struct{
    ppu_add_t buffer_cb [NUM_BUFS];
    atomic_ea_t valid [NUM_BUFS];
    atomic_ea_t valid_addr[NUM_BUFS];
    unsigned int fill [NUM_BUFS];
}spu_cb_t;
```

The `buffer_cb` parameter represents the address in Main Memory where a task buffer allocated for the SPU buffer being processed is located. The `valid` parameter indicates the validity of this buffer to the SPU SRE, allowing the SPU SRE to engage the processing of the task. Finally, the `valid_addr` parameter is simply the address of the equivalent `valid` signal kept by the PPU SRE. The `fill` parameter is of no consequence. Since this information is transferred across the MFC, it must follow the 16-Byte data-alignment requirement the MFC architecture specifies. The `fill` parameter is just used to complete this 16-byte requirement. The information described must be available for each buffer. It is specified during SPU start-up to setup the communication with the PPU SRE. Generally, it constitutes what will henceforth be named the SPU Control Block (SPU CB).

Once the SPU SRE has received the SPU CB, it can start processing tasks. To process each task, the SPU SRE needs 4 elements of information as explained previously when describing the task buffer:

1. the task ID
2. the input DMA list

3. the output DMA list

4. the address of a flag to signal writeback completion

The task ID is necessary for several reasons: it determines the sizes of the input and output DMA lists from a look-up table, and of course directs the SRE to run the appropriate task code.

The input and output DMA lists are naturally different for every task type launched in their structure. Further, given a certain task type, the addresses they point to will vary from task to task. Dedicated buffers are kept by the SPU SRE to hold both of a task's DMA lists so they may be used at the appropriate time.

The memory structure of a single SPU SRE buffer is described by Figure 4.5.

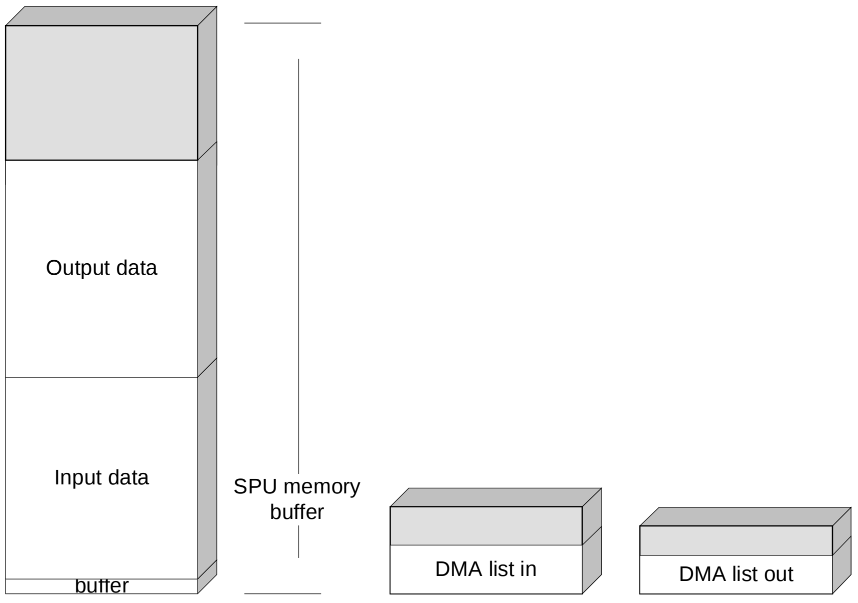

Fig. 4.5: The memory structures to hold task buffering information.

something else

4.4 PPU anatomy

The SPU SRE is in charge of performing the data transfers and setting up the execution of tasks on the SPUs. These tasks are created, coordinated, dispatched and managed by the PPU SRE. Functionally, the responsibilities of the PPU SRE can be divided in two parts. First, it must assign tasks to SPU task buffers when they are free and that the data for these tasks is ready. Second, it must advance program execution thereby generating and destroying tasks and supertasks as they are first needed then completed. Therefore, from a functional point of view, the PPU SRE must on the one side keep an updated picture of the activity on the SPUs. On the other hand, it must track the completion of tasks on the SPUs in order to notice the availability of new data to be processed by further tasks and supertasks.

Several structures are maintained by the PPU SRE to implement this functionality. To track the availability of SPU task buffers, the Dispatch Table keeps track of the SPU task buffers to which new tasks may be allocated. Given that every task is the child of a supertask, available tasks can be extracted by parsing a list of all active supertasks, the Supertask List. In order to update the status execution, completed tasks must notify their parent. To notice task completion, launched tasks are added to a Task List. This Task list is parsed by the PPU SRE in order to trigger the completed task to notify its parent of its completion.

4.4.1 Dispatch Table

To keep track of SPU activity, the SRE maintains a table of SPU buffers as depicted in Table 4.1. The elements in this table match exactly those of the SPU task buffers described in the previous Section, but holds this information for all SPUs' buffers. A new task can be assigned to this table by the PPU SRE as soon as the `valid` parameter indicates so. When a task is ready and the `valid` signal is de-asserted, the PPU SRE will assign the task to the buffer in the table. To do so, the task buffer information is fetched from the task and pointed to by the `buffer_cb` parameter. The `valid` parameter is asserted to indicate to the SPU SRE it may download the task buffer and start the processing of the task. The SPU SRE will de-assert the `valid` signal as soon as it has registered that the data necessary to execute the task is available. Therefore, the `valid` signal does not indicate *completion* of the task, as computation and writeback still have not occurred, but rather the possibility of assigning a new task to the given SPU buffer.

SPU	buffer	valid	buffer_cb_t	valid_address
0	0	1	&Iteration4:block 22	0xAA
0	1	0	NULL	0xBB
1	0	0	&Iteration4:block 24	0xCC
1	1	0	NULL	0xDD
2	0	0	&Iteration4:block 25	0xEE
2	1	0	NULL	0xFF

Table 4.1: A half active 3-SPU table with 2 buffers per SPU.

4.4.2 The Task List

As the Dispatch Table tracks the availability of SPU buffers, the Task List records active tasks. It is simply a linked list of tasks launched, as depicted by Figure 4.6. The labels inside the elements describe the iteration number and task within it for the Pease FFT example. This list is sweeped regularly to check upon task completion with the intent of signalling the parent supertask. This signalling will trigger the parent supertask to advance program execution given the newfound availability of the data computed by the task. Notice that this check is not equivalent to that of Table 4.1 since it checks upon completion of the writeback as opposed to the availability of the buffer (*c.f.* Section 4.5.3).

4.4.3 Supertask list

In much the same way that the Task List keeps track of active tasks, a Supertask List holds a list of active supertasks. The supertask list is yet another linked list that strings together

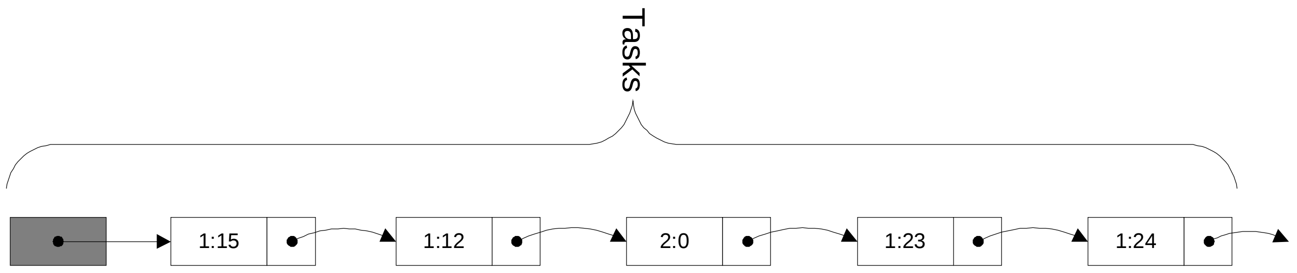

Fig. 4.6: The Task list is a single-pointer linked list of launched tasks.

active supertasks. The supertasks in the list are checked not only to verify completion, but rather also to fetch available tasks therefrom. Indeed, any supertask may be a source of tasks; as these become ready for dispatch, the parent supertask stores them in an internal list. *The order in which the supertask list is formed and read is therefore highly consequential on the the order in which tasks are dispatched.*

Figure 4.7 displays an example state of the supertask list during a certain point in execution. The Pease and LogIter supertasks represent the Pease function, and the second supertask of our design. The LogIter supertask has generated 3 iteration supertasks. Each of these have in turn queued several tasks for dispatch.

Extracting tasks from the Supertask List bears an interesting consequence. The relationship between two given supertasks is not necessarily hierarchical. Since a supertask may have more than one child supertasks, two supertasks may be sisters such as `iteration 0,1` and `2` in Figure 4.7. If the data dependency between these two supertasks permits it, two different branches of the DAG formed by Squid can be processed *simultaneously*. This behavior is a direct consequence of the dataflow model we have espoused and may lead to performance improvements that will be discussed in Section 4.5.2.

4.4.4 The Task Object

In this dynamic flow implementation, tasks and supertasks direct the flow of computation. They are implemented as C++ objects that receive and distribute data as it becomes avail-

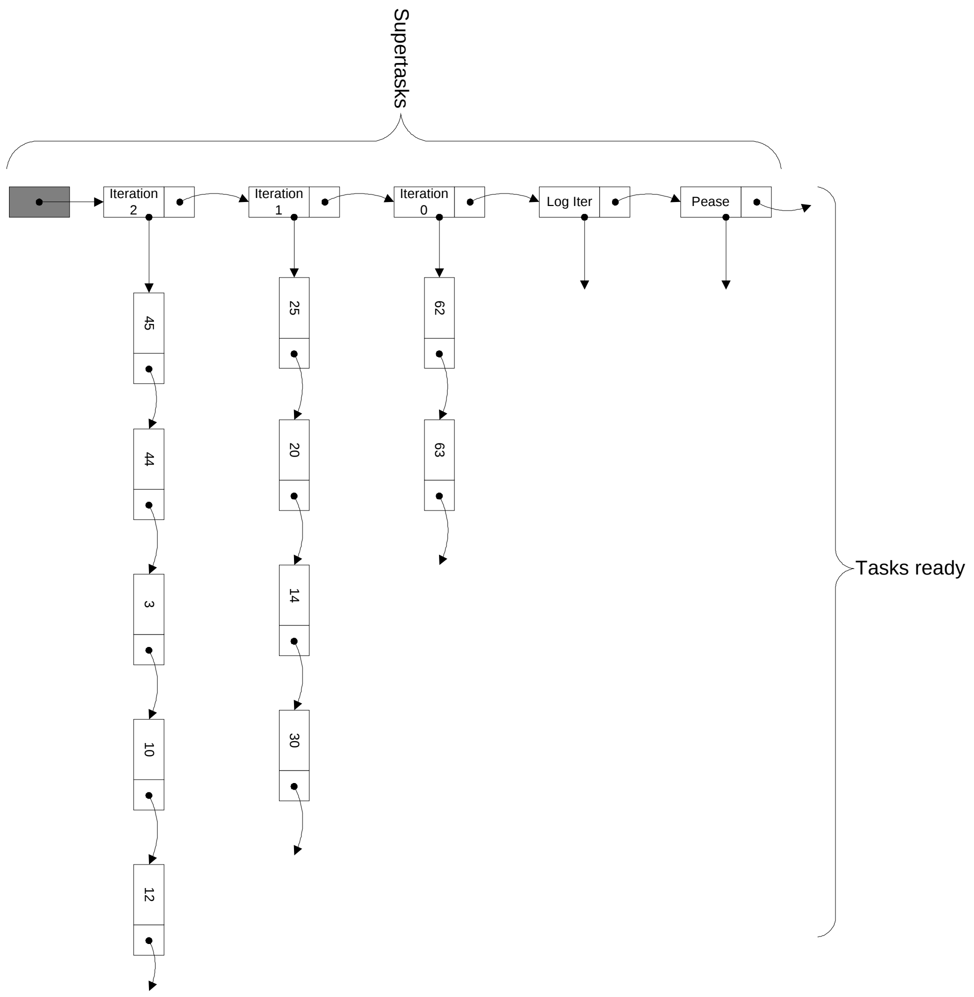

Fig. 4.7: The Supertask List is a linked list of supertasks where available tasks associated to each element of the list.

able.

Tasks are executed on the SPUs but some construct must exist in the PPU SRE so it may link to the appropriate SPU task buffer. Therefore, the PPU counterpart to the SPU tasks can only be a *wrapper* that act as a tunnel between their dataflow block structure on the PPU SRE and their computational representation –code– already stored on the SPU.

The task objects are fed inputs, and must send back outputs. Therefore, a task object must hold `set()` and `get()` functions for the parent supertask to perform these actions. It must also notify its parent supertask when all its inputs are available so it becomes 'hot' and may be appended at the end (or beginning) of the supertask's available tasks list.

Most importantly, the task object must contain a `checkTask()` function. This function is called by the SRE to check on task completion (*c.f.* Section 4.5.2). The SPU's buffer described in Section 4.3 holds the address of the `status` parameter of the task object. When the `checkTask()` function is called by the SRE, it checks locally for task completion. In the affirmative, it notifies the parent supertask of the availability of its outputs. Otherwise, the function returns notifying the SRE that the computation isn't finished.

4.4.5 The Supertask Object

Supertask object anatomy is more complex than its task object counterpart. First, it must not only point to available data, but in fact hold it. Further, it is responsible for directing dataflow to its child tasks and supertasks. Finally, it holds and maintains a list of available tasks.

A supertask may wrap around more than a single child. Internal dependencies and relationships between a supertask's children are managed by the parent. It is in the parent's hands that lies the responsibility of directing this flow. Therefore, in the case of the 'for' loop, the dependencies between the supertask's children are implicitly linked to the loop index they carry. Thus, in the case of the presence of an LCD, the same dependency tracking mechanism applies. The parent supertask passes as an argument an incremented loop index when passing one iteration's result as input to the next.

As it is apparent from the discussion in Chapter 3, there is a certain number of possible supertasks. Indeed, these may represent parallel 'for' loops, 'while' loops, 'for' loops with LCD's, 'if-else' conditional blocks, and 'switch-case' statements. In all these cases, supertasks should offer the same functionality to the SRE. So far, only the two kinds of 'for' loops have been implemented. However, these present the most complex design challenge, and provide a sound basis for the development of the rest.

4.4.6 The Data Object

Quite similarly to the concept of a function's stack used by `gcc`, data created outside the inner scope of a task is stored in the supertask for which it is created. In the case where the task output is part of a collection in the overarching supertask, the latter should hold the data. Revisiting our example:

```
// outer loop:
// top-level Supertask: LogIter
Z = for(i in [0 ..LOG_DATA-1]){
    // inner loop:
    // second tier supertask: Iteration
    Z_interim = for(j in [0 .. 63]){
        //task level:
        Z_task = for(k in [0 .. 1023]){
            z = ...
        }(<> z);
        // end of task
    }(><Z_task);
    // end of Supertask: Iteration
}Z_interim;
// end of Supertask: LogIter
```

where both `z` and `Z_task` are subsets of `Z_interim`, itself a subset of `Z`. In all these cases, the location of the data object for every iteration is stored at the level of the `LogIter` supertask. The other supertasks are thus passed by reference the members of the highest-level data element `Z`.

Since data is created dynamically, records must be kept of its availability. Indeed, one supertask may need, for example, several consecutive blocks of an array variable, only some of which are ready. Therefore, for tasks and supertasks alike, every data element has an associated a marker of its availability. This marker is asserted when a data element is signalled as available.

An important consideration in all object-oriented programming is garbage collection. As Tasks and Supertasks are destroyed, the data they handle must be destroyed as well. However, since several of these may be active at the same time, the amount of memory managed may be needlessly large. Clearly, some data will not be required once it is consumed, and its data may be destroyed at once, freeing up memory. The observations made to justify the dynamic allocation of memory for data applies in the reverse case to some extent. If a data block is referred to by affine indexing, *the amount of such references is the same at runtime.* Therefore, as Squid divides data blocks through its study of the references made to them, simply counting the numbers of such references above the task level is sufficient to determine at runtime the moment to destroy the data. A simple decrementer counts the number of tasks that called the data block. It is destroyed when the decrementer reaches zero. If the reference to the variable is not affine, it is destroyed at the same moment as its parent supertask.

4.5 Mechanics of the SRE

Several aspects of the dynamics of the SRE were suggested through the description of the SRE's anatomy. However, an in-depth look at the algorithms and intricacies of the management of so many concurrent processes will offer more insight than will a simple description of the SRE's components.

4.5.1 SPU SRE

The operation of SRE on the SPU is relatively simple as depicted by the flow chart on Figure 4.8. The first and foremost purpose of the SPU SRE is to overlap execution with data transfers. It must also coordinate with the PPU the availability of the buffers and the completion of the tasks.

Naturally, the first step the SPU SRE must take is to download the SPU CB, as it holds the information on the location for all the buffers. The address of the SPU CB is passed as a parameter to the top-level SPU code.

Once the SPU CB is downloaded, the SPU can start working on the tasks attributed to it

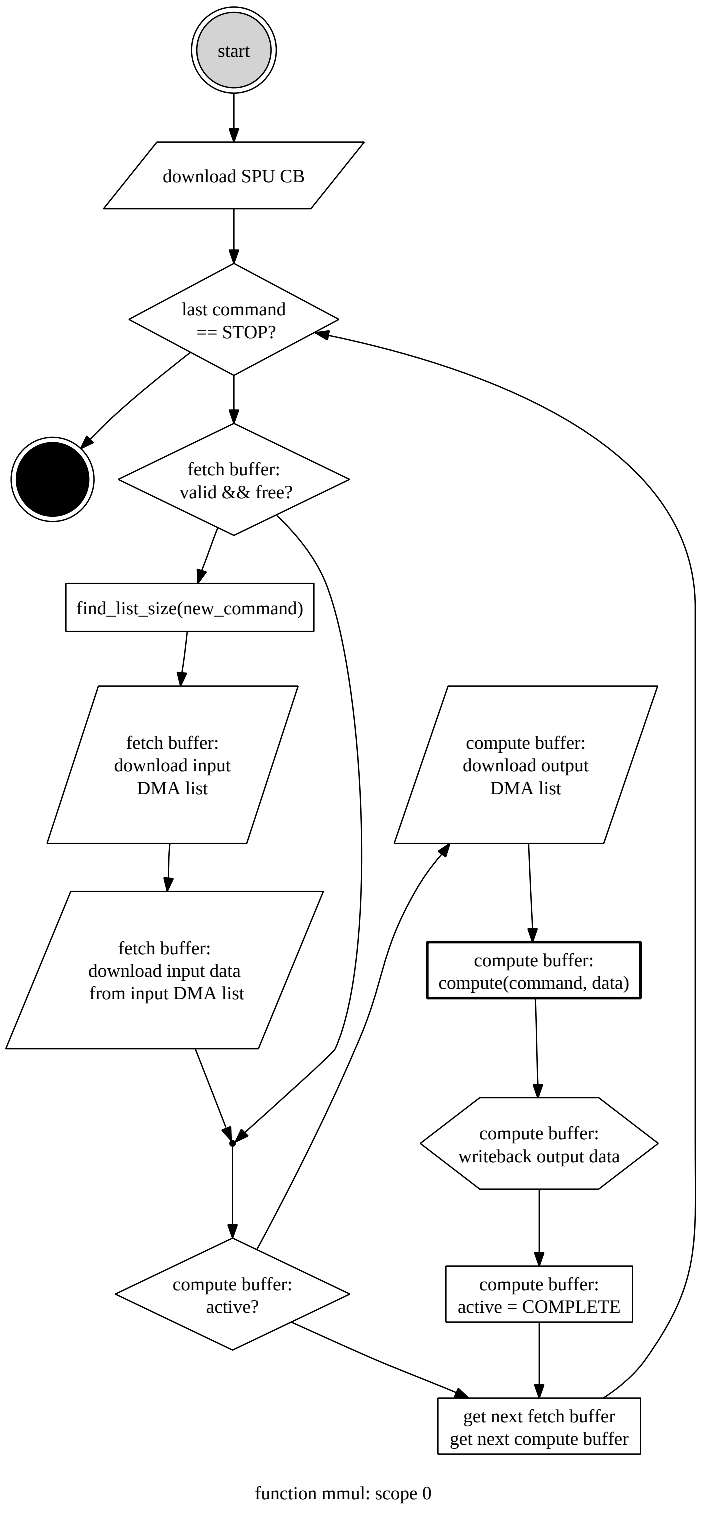
start
download SPU CB
last command == STOP?
fetch buffer: valid && free?
find_list_size(new_command)
fetch buffer: download input DMA list
fetch buffer: download input data from input DMA list
compute buffer: download output DMA list
compute buffer: compute(command, data)
compute buffer: writeback output data
compute buffer: active?
compute buffer: active = COMPLETE
get next fetch buffer get next compute buffer
function mmul: scope 0

Fig. 4.8: Flow chart of the operation of the SPU SRE.

by the PPU SRE. The SPU SRE performs three distinct functions:

- fetching input data
- computing
- writeback

Structurally, the mechanism of the SPU SRE consists of a single thread of code that iterates until it encounters a command to stop. It is constituted of two parts: the first to fetch input data, and the second to compute the task at hand. Data writeback is embedded in the computing step, but its execution overlaps the data fetch of the task to come.

As a first step, the SPU SRE attempts to fetch the input DMA list and the input data it represents. Starting with the base buffer (there could be two or more), the SPU SRE checks that a task is assigned by the PPU and that a new task may be downloaded to this location. In the affirmative, the SPU SRE proceeds to download the appropriate buffer. Since the buffer is quite small (16 bytes), the SPU SRE synchronizes on its completion to start downloading the input DMA list. The size of the list is directly dependent on the type of task to compute, the identification of which was passed as part of the buffer. Although this transfer is usually larger than the 16 Bytes of the buffer, it is usually small enough that it is worth wasting a bit of time synchronizing on its download as well. All that remains is to queue up the DMA list transfer to the appropriate memory buffer.

As a second step, the SPU SRE seeks to compute a task that is done downloading its inputs. Before proceeding to the computation, it first downloads the output DMA list in order to overlap the data transfer with the execution. Then, given the availability of the task ID that was stored at the bottom of the task buffer during the fetch stage, the appropriate task is executed. Since the DMA list performs a `gather` operation, all the data fetched is contiguous. A single base address for the input can thus be passed as an argument to the function corresponding to the task at hand. Finally, once the computation is completed, the output data is written back to Main Memory by means of the output DMA list fetched prior to the start of execution.

The astute observer might be concerned that there is no synchronization between the writeback of output data and the fetching of new data. Indeed, the MFC could potentially order them in the opposite way, or simply process both simultaneously. As a result, part or all of the data just loaded will overwrite that just computed in the midst of the writeback. For this reason, the data transfers are ordered by means of a *fence* that forces the `getl()` command to follow the writeback. Input and output data are thereby ordered with respect to one another, and there is no need to force the data fetch step to wait for completion of the writeback.

4.5.2 PPU SRE

The PPU SRE consists as well of two functions executed on a loop: `update()` and `dispatch()`. The first checks upon every element of the Task List to observe for task completion. If a task has indeed completed its writeback, the task will trigger a cascade of function calls that advance program execution. Indeed, every task has functions dedicated to notifying its parent supertask of the availability of its outputs. The parent supertask, in turn, 'binds' to this call to update its knowledge of valid data blocks. It follows the dependencies deciphered by Squid to notify in turn the child tasks and/or supertasks of the available data.

In this chain of notifications, tasks and supertasks are created and destroyed. Indeed, in order to optimize memory use, tasks and supertasks are not instantiated until the moment they are required. Thus, they are created when and only when data dedicated to them is notified as available. In the case where only some of a task's inputs are available upon creation, in no way should the task be added to the supertask list of available tasks. It should however be added as soon as the task notices all its inputs as available.

Referring once more to our example, every task requires two inputs. These inputs are not available at the same time: a task creates two adjacent quantas of `Z`, and a task's input blocks of the previous iteration of `Z` are separated by a distance of half the total number of

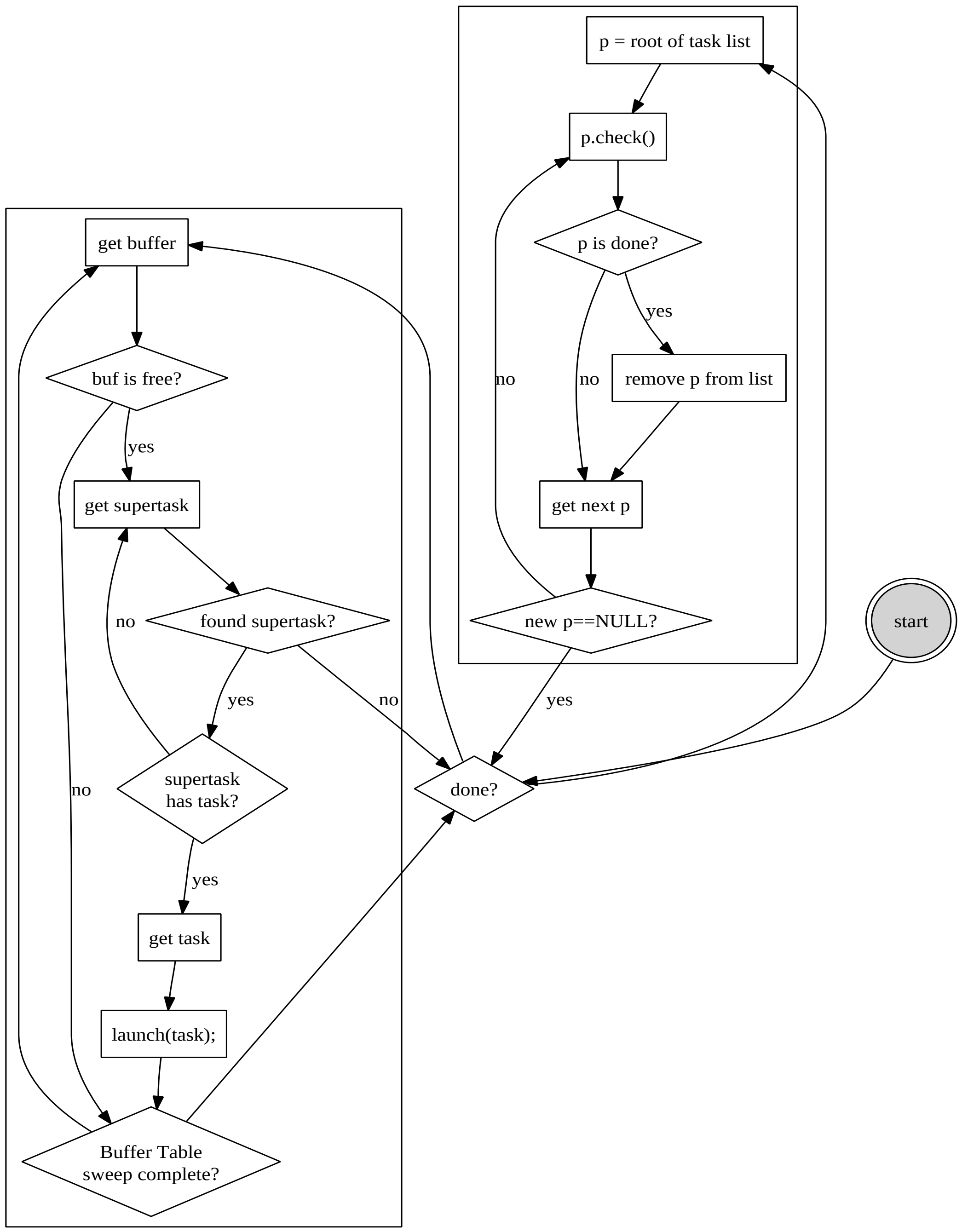

Fig. 4.9: High-level flowchart of PPU SRE execution

blocks:

- Z[j]

- Z[j+SIZE/2]

Therefore, both blocks are not available simultaneously. The task is ready to be disptched only once both are available.

The new tasks and supertasks are respectively added to the parent supertask's list of available tasks and the supertask list. The way in which the former are added to the latter is certainly consequential to the order of program execution as previously explained. This observation could have an effect on performance. As the time between some data's writeback and new dispatch elapses, the physical memory location of the data moves from cache to Main Memory to physical memory. The farther the location of the data, the lengthier the memory fetch time and the more application time is lost. Therefore, there is clearly an advantage to consume data as quickly as possible once it is produced. The order in which the Supertask List is constructed and parsed can thus have an impact on performance. In our example, once a task's second input is just sent back, it will trigger the notification of a newly available task. The sooner the new task is dispatched, the more likely its inputs to be fetched quickly, particularly in the data just sent back.

It is in the `dispatch()` part of the SRE execution cycle that the Supertask list is parsed. The SRE checks for free buffers in the Dispatch table described in 4.4.1. When a buffer is indeed free, a new task will be assigned to the buffer. This task can be a child of any active supertask in the Supertask List. The first supertask to be looked at for ready tasks is typically the root of the supertask list. Hence, adding supertasks by appending them to the supertask list will lead to a breadth-first search. In the case of a loop containing a very large parallel number of tasks, one task's output will be consumed only when all the others have been produced. At this point, it is quite likely that the data will be relegated quite far down the memory hierarchy, and every task's output will require a long time to fetch. In the opposite case, however, a depth-first search occurs, taking full advantage of data temporal and physical locality.

4.5.3 Coordinating Between the PPU and SPU SRE's

The low-level coordination between the PPU and SPU SRE's is depicted in Figure 4.10. The `valid` and `status` buffer parameter are the two flags that are needed for this coordination. The SPU SRE checks the `status` signal in order to attempt the processing of the task associated to the buffer. If it is asserted, a task is ready for processing, and the mechanism described by Figure 4.8 is engaged. As the SPU SRE starts its buffer fetch phase for a given buffer, this buffer's `valid` signal is de-asserted in order to indicate to the PPU SRE to dedicate a new task for the buffer. When the PPU SRE parses its Dispatch Table, the de-asserted `valid` signal provides it an indication that a new task may be attributed to it, even if the computation is far from complete. When it attributes the new task to the buffer, the `valid` signal is re-asserted.

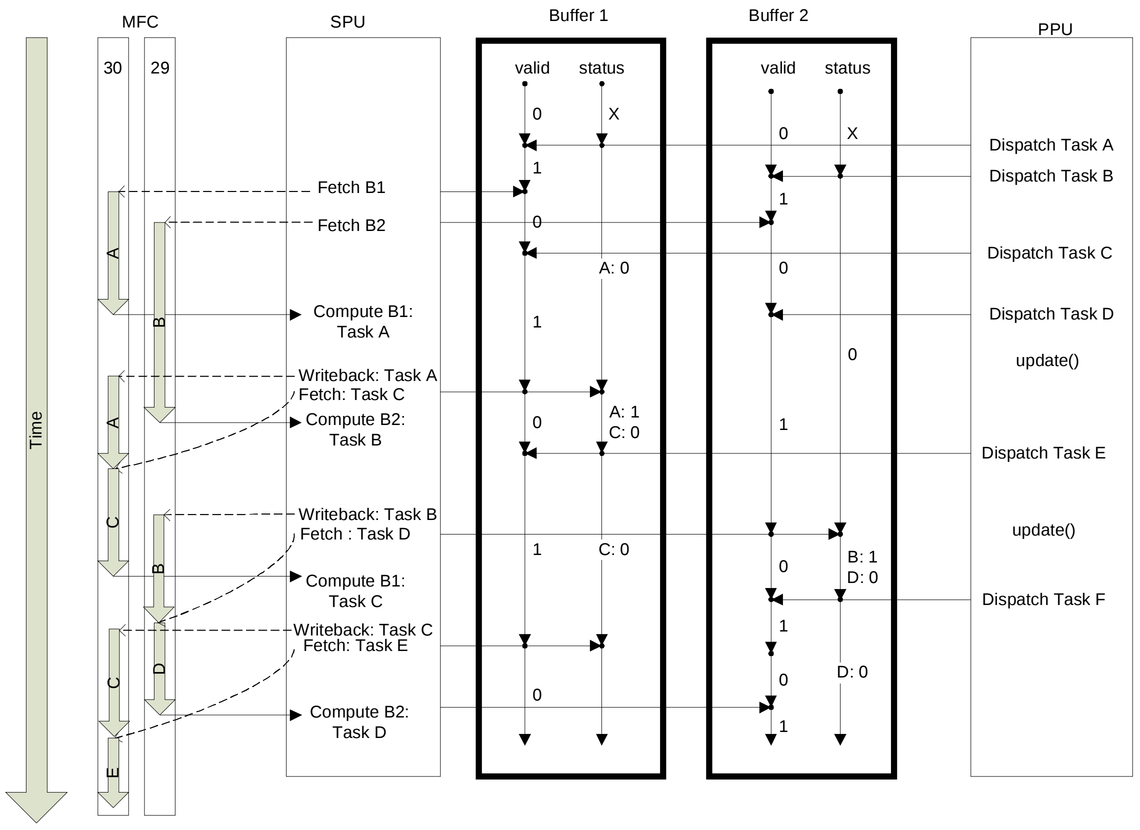

Fig. 4.10: Coordinating through the MFC between the PPU SRE and the SPU SRE.

The `status` signal indicates buffer completion, and is proper to the task object it represents. It is asserted when the task has finished its writeback. The synchronization on the writeback of a task on a given buffer is performed as a new task is about to start its computation phase for the same buffer. Typically, it is at this point that the `status` signal is asserted.

Updating the `status` signal only in this situation causes a risk. A DAG may happen to be 'narrow' at a given point, causing new tasks' readiness to depend on the last computed. However, if no new task is dispatched, the first task's `status` signal is never asserted causing a deadlock! For this reason, as the SPU SRE iterates over the buffers, it verifies writeback completion as well as availability of a new task.

4.6 Simple Results

Unfortunately, the testing we performed was rudinmentary. It consists of two implementations of the Pease FFT: the first uses the technique we applied, and the second was made slower on purpose. Running trial tests, we quickly realized that the 2x1024-element tasks we described previously completed too quickly to show a significant speedup. We then proceeded to force each task to compute every twiddle factor, requiring much more time. This generated the results of Table 4.2.

Number of SPUs	1	2	3	4	5	6
slow	1	2	2.94	4	4.76	5.88
fast	1	2.2	2.55	3.62	4.79	5.53

Table 4.2: Speedup of our Pease FFT implmentations with respect to a one-SPU performance

These results show a virtully linear speedup in both cases. However, one can see that the speedup factor for the fast kernel is moderately smaller than in the case of the slow kernel. There are three possible reasons for this:

- The computation kernel is too fast with respect to the time required to download data, in spite of multiple-buffering techniques.
- As the number of active SPE's increases, the amount of traffic that accesses Main memory creates a logjam and slows overall progression.
- As the number of active buffers increases, the SRE may take too much time in assigning buffer use in real time.

The cause of the reduction in speedup between different kernels can be caused by any of these factors. In this case, it is likely to be due to the combined effect of all three. More tests and different applications are necessary to study the impact of these factors.

4.7 Future Work

The structures and mechanisms of the SRE are well contained and coherent. Yet, several aspects of its structures and mechanisms may be remodeled and re-examined. Beyond the relatively simple case of the Pease FFT algorithm, a benchmark suite is clearly a top priority to analyze more thoroughly the inefficencies in the system and possible improvements. Very justifiable questions may be raised as to the load of management and book-keeping placed on the PPU. Similarly, the SPU SRE may spend too much effort on task management, some of which may be performed by the PPU. The trade-off in the amount of management load between the PPU and SPU SREs is thus quite important. We believe the approach taken is reasonable however, and migrating the load from one SRE to the next should be relatively straightforward to apply.

Chapter 5

Conclusion

In Conclusion, we have presented a novel integrated solution to programming on the CBE that could extend well to other mutli-core architectures. As a first step, we introduced the NCC language, a strict functional language that explicits data dependencies. The Squid Compiler then parses and analyzies the code to extract data dependencies and partition the code using a maximum-fit technique. Divided code is translated to C to figure as tasks on SPUs. Supertasks direct the flow of execution of tasks by enforcing data dependencies during runtime by communicating with the SRE. The SRE is itself responsible for the dynamic distribution of tasks, enforcing thread-level data coherency. In order to optimize the computational power of the SPUs, multiple-buffering is used to overlap the arrival of a task's input data and the computation of that data.

Despite its promise, our solution still requires significant testing using intelligent banchmarks. We have left this as future work, and intend to pursue further optimizations using advanced techniques not yet implemented.

References

[1] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick, "The potential of the cell processor for scientific computing," in *Proceedings of the 3rd conference on Computing frontiers.* ACM New York, NY, USA, 2006, pp. 9–20.

[2] Intel, "Teraflops resarch chip." [Online]. Available: http://techresearch.intel.com/articles/Tera-Scale/1449.htm

[3] J. Davis, J. Laudon, and K. Olukotun, "Maximizing cmp throughput with mediocre cores," *Parallel Architectures and Compilation Techniques, 2005. PACT 2005. 14th International Conference on*, pp. 51–62, Sept. 2005.

[4] P. Kongetira, K. Aingaran, and K. Olukotun, "Niagara: A 32-Way Multithreaded Sparc Processor," *IEEE MICRO*, pp. 21–29, 2005.

[5] K. Asanovic, R. Bodik, B. Catanzaro, J. Gebis, P. Husbands, K. Keutzer, D. Patterson, W. Plishker, J. Shalf, S. Williams, *et al.*, "The Landscape of Parallel Computing Research: A View from Berkeley," *Electrical Engineering and Computer Sciences, University of California at Berkeley, Technical Report No. UCB/EECS-2006-183, December*, vol. 18, no. 2006-183, p. 19, 2006.

[6] A. Chandrakasan, S. Sheng, and R. Brodersen, "Low-power CMOS digital design," *Solid-State Circuits, IEEE Journal of*, vol. 27, no. 4, pp. 473–484, 1992.

[7] R. Kumar, D. M. Tullsen, P. Ranganathan, N. P. Jouppi, and K. I. Farkas, "Single-isa heterogeneous multi-core architectures for multithreaded workload performance," *SIGARCH Comput. Archit. News*, vol. 32, no. 2, p. 64, 2004.

[8] G. Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," in *AFIPS Conference Proceedings*, vol. 30, no. 8, 1967, pp. 483–485.

[9] J. B. Andrews and C. D. Polychronopoulos, "An analytical approach to performance/-cost modeling of parallel computers," *J. Parallel Distrib. Comput.*, vol. 12, no. 4, pp. 343–356, 1991.

[10] M. Becchi and P. Crowley, "Dynamic thread assignment on heterogeneous multiprocessor architectures," in *CF '06: Proceedings of the 3rd conference on Computing frontiers.* New York, NY, USA: ACM, 2006, pp. 29–40.

[11] T. Sondag, V. Krishnamurthy, and H. Rajan, "Predictive thread-to-core assignment on a heterogeneous multi-core processor," in *PLOS '07: Proceedings of the 4th workshop on Programming languages and operating systems.* New York, NY, USA: ACM, 2007, pp. 1–5.

[12] S. IBM, *Cell Broadband Engine Architecture*, 2006.

[13] J. Wetzel and IBM, *User Instriction Set Architecture, books I, II, III*, 2005.

[14] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Interconnection Network: Built for Speed," *IEEE Micro*, vol. 26, no. 3, 2006.

[15] S. Alam, J. Meredith, and J. Vetter, "Balancing Productivity and Performance on the Cell Broadband Engine," in *Proc. of the 2007 IEEE Annual International Conference on Cluster Computing, September*, 2007.

[16] IBM, *Software Development Kit for Multicore Acceleration, version 3.0: Programming tutorial*, 2007.

[17] *Cell Broadband Engine Programming Handbook*, 2007.

[18] A. Varbanescu, H. Sips, K. Ross, Q. Liu, L. Liu, A. Natsev, and J. Smith, "An Effective Strategy for Porting C++ Applications on Cell," in *Parallel Processing, 2007. ICPP 2007. International Conference on*, 2007, pp. 59–59.

[19] A. Eichenberger, K. OBrien, K. OBrien, P. Wu, T. Chen, P. Oden, D. Prener, J. Shepherd, B. So, Z. Sura, *et al.*, "Optimizing Compiler for a CELL Processor."

[20] IBM, *SPE Runtime Management Library*, 2007.

[21] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani, "MPI microtask for programming the Cell Broadband Engine processor," *IBM Systems Journal*, vol. 45, no. 1, pp. 85–102, 2006.

[22] K. OBrien, K. OBrien, Z. Sura, T. Chen, and T. Zhang, "Supporting OpenMP on Cell," *International Journal of Parallel Programming*, vol. 36, no. 3, pp. 289–311, 2008.

[23] G. Sorst and M. Deuling, "Diploma Thesis: Java on Cell BE."

[24] I. Inc., *ALF for Hybrid-x86 Programmers Guide and API Reference*, 2007.

[25] Y. Zhao and K. Kennedy, "Dependence-Based Code Generation for a CELL Processor," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4382, p. 64, 2007.

[26] R. Allen and K. Kennedy, "Vector register allocation," *IEEE Transactions on Computers*, vol. 41, no. 10, pp. 1290–1317, 1992.

[27] IBM, *IBM Software Development Kit for Multicore Acceleration v3.0: Basic Linear Algebra Subprograms Programmers Guide and API Reference*, 2007.

[28] M. Frigo and S. Johnson, "FFTW: an adaptive software architecture for the FFT," in *Acoustics, Speech, and Signal Processing, 1998. ICASSP'98. Proceedings of the 1998 IEEE International Conference on*, vol. 3, 1998.

[29] M. Frigo and S. Johnson, "FFTW on the Cell Processor." [Online]. Available: http://www.fftw.org/cell/

[30] F. Blagojevic, D. Nikolopoulos, A. Stamatakis, C. Antonopoulos, and M. Curtis-Maury, “Runtime scheduling of dynamic parallelism on accelerator-based multi-core systems,” *Parallel Computing*, 2007.

[31] F. Blagojevic, D. Nikolopoulos, A. Stamatakis, and C. Antonopoulos, “Dynamic multi-grain parallelization on the cell broadband engine,” in *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming.* ACM Press New York, NY, USA, 2007, pp. 90–100.

[32] A. Grama, G. Karypis, V. Kumar, and A. Gupta, *Introduction to Parallel Computing.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2002.

[33] C. Koelbel, *The High Performance Fortran Handbook.* MIT Press, 1994.

[34] R. Chandra, *Parallel Programming in OpenMP.* Morgan Kaufmann, 2001.

[35] M. Snir, S. Otto, D. Walker, J. Dongarra, and S. Huss-Lederman, *MPI: The Complete Reference.* MIT Press Cambridge, MA, USA, 1995.

[36] L. Hochstein, J. Carver, F. Shull, S. Asgari, and V. Basili, “Parallel programmer productivity: A case study of novice parallel programmers,” in *SC ’05: Proceedings of the 2005 ACM/IEEE conference on Supercomputing.* Washington, DC, USA: IEEE Computer Society, 2005, p. 35.

[37] S. Asgari, V. Basili, J. Carver, L. Hochstein, J. Hollingsworth, F. Shull, and M. Zelkowitz, “Challenges in Measuring HPCS Learner Productivity in an Age of Ubiquitous Computing: The HPCS Program,” in *Proceedings of ICSE Workshop on High Productivity Computing*, 2004, pp. 27–31.

[38] L. Lamport, “The parallel execution of DO loops,” *Communications of the ACM*, vol. 17, no. 2, pp. 83–93, 1974.

[39] *Compilers: principles, techniques and tools.* Pearson Education Inc., 2007.

[40] G. Dantzig, “Fourier-motzkin elimination and its dual.” DTIC, Tech. Rep., 1972.

[41] D. Michie, “Memo functions and machine learning,” *Nature*, vol. 218, no. 1, pp. 19–22, 1968.

[42] H. Barendregt, “The Lambda Calculus, its Syntax and Semantics, Revised second edition,” 1984.

[43] J. Hughes, “Standard haskell.” [Online]. Available: www.cs.chalmers.se/r̃jmh/Haskell

[44] H.-W. Loidl, “Granularity in large-scale parallel functional programming,” Ph.D. dissertation, University of Glasgow, March 1998. [Online]. Available: http://www.dcs.gla.ac.uk/ hwloidl/publications/PhD.ps.gz

[45] J. T. Feo, D. C. Cann, and R. R. Oldehoeft, “A report on the sisal language project,” *J. Parallel Distrib. Comput.*, vol. 10, no. 4, pp. 349–366, 1990.

[46] R. Nikhil, “Id Language Reference Manual Version 90.1,” *Computation struc*, 1991.

[47] S. Aditya, L. Arvind, J. Maessen, and R. Nikhil, “Semantics of pH: A parallel dialect of Haskell,” in *Haskell Workshop*, 1995.

[48] L. Kale, “Performance and productivity in parallel programming via processor virtualization,” in *Proc. of the First Intl. Workshop on Productivity and Performance in High-End Computing (at HPCA 10)*, 2004.

[49] I. Buck and e. a. Foley, T., “Brook language,” 2007. [Online]. Available: http://graphics.stanford.edu/projects/brookgpu/lang.html

[50] N. Inc., *NVIDIA CUDA Compute Unified Device Architecture: Programming Guide*, 2007.

[51] P. Wadler, “Why no one uses functional languages,” *ACM SIGPLAN Notices*, vol. 33, no. 8, pp. 23–27, 1998.

[52] E. Inc., *Erlang/OTP R12B*, 2007.

[53] “Projectfortress community.” [Online]. Available: http://projectfortress.sun.com/Projects/Community

[54] M. Inc., “The mitrion sdk.” [Online]. Available: http://mitrionics.com/

[55] J. Cooley and J. Tukey, “An algorithm for the machine calculation of complex Fourier series,” *Math. Comput*, vol. 19, no. 90, pp. 297–301, 1965.

[56] A. Ali, L. Johnsson, and J. Subhlok, “Scheduling fft computation on smp and multicore systems,” in *ICS ’07: Proceedings of the 21st annual international conference on Supercomputing.* New York, NY, USA: ACM, 2007, pp. 293–301.

[57] D. Bader and V. Agarwal, “FFTC: Fastest Fourier Transform for the IBM Cell Broadband Engine,” *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4873, p. 172, 2007.

[58] M. C. Pease, “An adaptation of the fast fourier transform for parallel processing,” *J. ACM*, vol. 15, no. 2, pp. 252–264, 1968.