SoC-C: Efficient Programming Abstractions for Heterogeneous Multicore Systems on Chip

Alastair D. Reid
Krisztian Flautner
Edmund Grimley-Evans
ARM Ltd

Yuan Lin
University of Michigan

ABSTRACT

The architectures of system-on-chip (SoC) platforms found in high-end consumer devices are getting more and more complex as designers strive to deliver increasingly compute-intensive applications on near-constant energy budgets. Workloads running on these platforms require the exploitation of heterogeneous parallelism and increasingly irregular memory hierarchies. The conventional approach to programming such hardware is very low-level but this yields software which is intimately and inseparably tied to the details of the platform it was originally designed for, limiting the software's portability, and, ultimately, the architectural choices available to designers of future platform generations. The key insight of this paper is that many of the problems experienced in mapping applications onto SoC platforms come not from deciding how to map a program onto the hardware but from the need to restructure the program and the number of interdependencies introduced in the process of implementing those decisions. We tackle this complexity with a set of language extensions which allows the programmer to introduce pipeline parallelism into sequential programs, manage distributed memories, and express the desired mapping of tasks to resources. The compiler takes care of the complex, error-prone details required to implement that mapping. We demonstrate the effectiveness of SoC-C and its compiler with a "software defined radio" example (the PHY layer of a Digital Video Broadcast receiver) achieving a 3.4x speedup on 4 cores.

Categories and Subject Descriptors: D.3.3 [Software]: Programming Languages

General Terms: Languages

1. INTRODUCTION

In the next five years the peak available bandwidth to mobile phones is expected to increase from less than 5 Mbps today to 100 Mbps in 2012. The signal-processing throughput to implement these protocols is expected to increase to beyond 25 giga-operations per second. Commodity cameras on phones already support 10M pixel resolution which further drives the need for high-speed multimedia image processing, high-definition video coding and 3D graphics. To maintain the same form-factor, this massive performance must be achieved without increasing battery size which limits the power consumption to around 1 Watt.

CASES'08, October 19–24, 2008, Atlanta, Georgia, USA.

Modern DSP designs are starting to achieve the required energy efficiency. For example, ARM's prototype data processing engine can sustain over 10 GMAC/s at less than 300mW in 65nm technology. *The main problem is not creating energy-efficient hardware but creating efficient, maintainable programs to run on them.* In order to save energy and, to some extent, silicon area, high performance embedded systems eschew features that characterize today's high-end multiprocessor systems: Homogeneous processors are replaced by a heterogeneous mix of specialized processors tuned to particular parts of the expected workload; General-purpose processors programmed in C, C++, etc. are supplemented by special-purpose accelerator engines which may be fixed-function, configurable or programmable using a C subset; Shared memory is replaced by multiple private memories to decrease latency and energy and increase bandwidth; and Hardware cache coherency is omitted to save area and power consumed by cache coherence protocols. Omitting these features from high performance embedded systems requires programmers to adopt a very low-level, error-prone programming style that limits portability and maintainability. *The key insight of this paper is that these problems come not from deciding how to map the application onto the hardware but from the restructuring of the code and the number of interdependencies introduced in the process of implementing those decisions.* Rather than abandon features because of their hardware cost, SoC-C moves their implementation into the language so that the programmer can reason about and optimize the mapping at a high level while the compiler takes care of the complex, error-prone details required to implement that mapping.

SoC-C is a set of language extensions that enables programmers to express efficient system-on-chip programs that exploit the parallelism available in the platform, provides programmers with control over how the many different processing elements in the platforms are used, and requires little or no restructuring when the application is subsequently ported within a family of platform architectures.

This paper makes the following contributions: We describe channel-based decoupling: a novel spin on existing ways to automatically introduce pipeline parallelism that allows programmers to tradeoff determinism for scheduling freedom and is capable of handling the complex control flow that real applications require. We propose a novel way of expressing the data copying that must happen in a distributed memory system. Our annotations express the programmer's intent allowing the compiler to detect missing or incorrect copy operations. We describe an inference mechanism that

```
// Data placement
declaration ::= type variable @ { memory1, ... memoryn } ;
expression  ::= variable @ memory
statement   ::= SYNC(variable[,memory[,memory]]
                        ) @ processor ;
// Code placement
expression  ::= identifier( expression, ... expression
                        ) @ processor
// Fork-join parallelism
statement   ::= parallel_sections {
                    section {  compound-statement } ;
                    ...
                    section {  compound-statement } ;
                }
// Pipeline parallelism
statement   ::= pipeline { compound-statement }
statement   ::= FIFO ( variable ) ;
```

Figure 1: SoC-C syntax extensions.

significantly reduces the amount of annotation required to map an application onto a hardware platform. We identify the critical optimizations required to support the high level programming model. With these optimizations, SoC-C can achieve accelerator utilization levels of 94% and a speedup of 3.4x on a platform with 4 accelerators on a real workload.

The paper is structured as follows. Section 2 describes a set of obvious minimal extensions to C to support heterogeneous, distributed parallel systems and introduces an example to illustrate why these extensions are *necessary* but *insufficient* for programming complex SoCs. Thus motivated, Sections 3–6 make a series of improvements showing how each extension improves the running example and we evaluate the expressiveness of the extensions in Section 7. Sections 8 and 9 discuss optimizations and performance. Section 10 discusses related work and Section 11 concludes.

This paper does not address how the best application mapping can be generated automatically using program analysis, profiling, iterative compilation, etc. for two reasons. The first is that the mechanism used to choose a mapping is largely orthogonal to the mechanism used to act on those decisions. The second is that there is no single obvious property to optimize for in embedded systems. Depending on the system one may want to optimize for some combination of battery life, low-latency user experience, meeting real-time deadlines, reducing number of retransmits, code size, etc.

2. A MINIMAL EXTENSION TO C

This Section considers minimal extensions to C to support heterogeneous multiprocessor systems with distributed memory and shows that whilst these or similar extensions are *necessary* (and form the basis of SoC-C), they are not *sufficient* for creating high performance, maintainable programs. This sets the stage for later sections which describe further extensions and optimizations to tackle these problems.

The extensions considered in this Section are those required to introduce parallelism, control sharing of resources and variables, communicate between threads, map data onto memories and map code onto processors/accelerators. Our descriptions of the extensions are brief because they are based on extensions found in other languages such as OpenMP (which inspired our notation), Concurrent Pascal, etc. Figure 1 summarizes all the extensions discussed in this paper.

Parallel sections introduce fork-join parallelism where a single master thread forks multiple child tasks (which may also fork child tasks) and waits for all children to complete.

```
complex_t samples[2048];
bool      bits[3024];
int8_t    bytes[378];
int       timing_correction = 0;
while (1) {
  ADC_get(&adc,&samples,2048);
  AdjustTiming(timing_correction,samples);
  FFT(samples);
  timing_correction += FindTimeOffset(samples);
  Demodulate(bits,samples);
  ErrorCorrect(bytes,bits);
}
```

Figure 2: A simplified OFDM radio receiver.

The statement

```
parallel_sections{
  section{ statement1 }
  section{ statement2 }
}
```

executes `statement1` and `statement2` in parallel and completes when both statements complete. Parallel sections can be implemented by forking one thread per section and then waiting for all threads to complete. Since this is the basic mechanism for expressing all parallelism, it is the programmer's responsibility to avoid race conditions, deadlock, etc.

Channels synchronize/communicate between threads. FIFO channels provide two operations: "`fifo_put`" atomically transfers data into the channel and "`fifo_get`" operations atomically transfers data out (blocking if the channel is full/empty). This atomic-transfer semantics ensures that each thread has exclusive access to the data.

Data placement annotations map variables to memories. A variable declaration of the form

```
type V @  M ;
```

instructs the SoC-C compiler and linker to place the variable '`V`' in memory '`M`'.

Code placement annotations perform RPCs. A function call of the form

```
function(expr1, ... exprm) @ P
```

is compiled into a synchronous remote procedure call: the function is invoked on processing element '`P`'. Unlike most RPC implementations, the call-frame (i.e., which function to call and any scalar and pointer arguments) is copied to the processing element but bulk data structures are not copied. This reflects our design goal of giving the programmer control over data copying to let them tune memory use and the impact on timing.

To illustrate these minimal extensions, consider mapping the sequential program in Figure 2 onto the architecture shown in Figure 3. This program displays two different types of data dependency which must be handled when parallelizing the program. There is forward dataflow within a loop iteration carrying complex samples from the ADC through timing correction, an FFT, demodulation and error correction. There is also feedback loop from one iteration to the next which continuously monitors changes in the timing offset between the transmitter and the receiver (caused by slight differences in clock rates, Doppler effects, etc.) which is used to control timing correction in future iterations. For simplicity, this example deals with fine timing correction (errors less than half the sample rate which are dealt with by applying a rotation to the complex samples) but ignores coarse timing correction (which would adjust the ADC in-

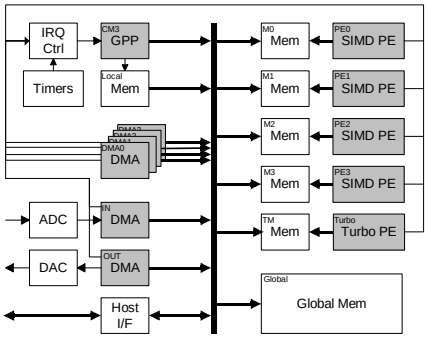

Figure 3: The architecture of a communication-processing subsystem. This system consists of one RISC processor and five processing elements (highlighted in gray), five private memories and one shared memory.

teraction) and channel equalization (which would correct for some frequencies being received more strongly than others).

In SoCs with heterogeneous processors, the principal form of parallelism used is pipeline parallelism: each engine is dedicated to performing a set of tasks and engines communicate with each other via FIFO channels. Figure 4 shows how the program can be rewritten to express pipeline parallelism using the minimal subset of SoC-C described in this Section. As can be seen, the parallel version of the program is significantly longer than the sequential version and has several more variables (both buffers and FIFOs). Looking more closely, we identify the following problems:

FIFO channels create excessive synchronization In the sequential version of the program, a feedback loop carries the timing correction back for use in the next iteration. To achieve exactly the same semantics, the parallel program would need to use a FIFO channel but this would have forced the first three sections to run sequentially because section 1 could not start the next iteration until section 3 had sent the new timing correction — a problem known as *loss of decoupling* [1]. To avoid this, the programmers used their knowledge of radio systems to confirm that timing corrections change slowly and so it would be acceptable to use a slightly older timing correction if that leads to greater parallelism. Since FIFO channels create too much synchronization, they chose the other inter-thread communication method of a shared variable accessed in a critical section. Section 3 addresses this by supporting user defined channels and zero-copy optimization.

Structure of the program is significantly changed. While the sequential program was a single, short loop, the parallel program contains four loops, the code is dominated by communication and parallelism constructs and it takes more effort to determine the flow of data through the program.

Equally seriously, this major restructuring was performed to suit one particular architecture and achieve a reasonable load balance given the current speed of each function. If the architecture were to change or a function were optimized, the program would have to be restructured accordingly — a significant and error-prone undertaking. Section 4 addresses this using decoupling to automatically introduce pipeline parallelism under programmer control.

Fragmentation of variables. Each individual variable in the sequential program has been fragmented into many separate variables in the parallel program. This fragmentation comes from two distinct sources: replicating variables that are communicated between threads and replicating variables between memory spaces. Whilst the replication is necessary, the burden on the programmer is significant: they may use the wrong version of a variable, they may fail to copy from one version of a variable to the other, or they may perform the copy but in the wrong direction. Sections 4 and 5 separately address the two sources of fragmentation.

```
int timing_correction = 0;
parallel_sections{
  section{
    complex_t samples1[2048] @ {M0};
    int       t1;
    while (1) {
      ADC_get(&adc,&samples1,2048);
      critical(offset){
        t1 = timing_correction;
      }
      AdjustTiming(t1,samples1) @ P0;
      fifo_put(&f1,samples1);
    }
  }
  section{
    complex_t samples2[2048] @ {M0};
    complex_t samples3[2048] @ {M1};
    while (1) {
      fifo_get(&f1,samples2);
      memcpy(samples3,samples2,sizeof(samples2)) @ DMA;
      FFT(samples3) @ P1;
      fifo_put(&f2,samples3);
    }
  }
  section{
    complex_t samples4[2048] @ {M1};
    complex_t samples5[2048] @ {M2};
    bool      bits1[3024]    @ {M2};
    int       t2 = 0;
    while (1) {
      fifo_get(&f2,samples4);
      memcpy(samples5,samples4,sizeof(samples4)) @ DMA;
      t2 += FindTimeOffset(samples5)@P2;
      critical(offset){
        timing_correction2 = t2;
      }
      Demodulate(bits1,samples5) @ P2;
      fifo_put(&f3,bits1);
    }
  }
  section{
    bool      bits2[3024] @ {M2};
    bool      bits3[3024] @ {M3};
    int8_t    bytes[378]  @ {M3};
    while (1) {
      fifo_get(&f3,bits2);
      memcpy(bits3,bits2,sizeof(bits2)) @ DMA;
      ErrorCorrect(bytes,bits3) @ Viterbi;
    }
  }
}
```

Figure 4: A parallel version of the program in Figure 2.

Performance issues In addition to the impact on the structure of the application, the choice of synchronous RPCs, threads and FIFOs to express parallelism is convenient but runs the risk of a high overhead from copying data and swapping thread contexts. Section 8 shows that existing optimizations can be used to allow the use of high-level constructs without excessive overhead.

3. USER DEFINED CHANNELS

In Section 2, we observed that FIFO channels introduced too much synchronization and therefore used shared vari-

```
typedef struct { lock_t lock; int data; } atomic_int_t;

void atomic_int_init(atomic_int_t *a, int x)
{ lock_init(&a->lock); a->data = x; }

void atomic_int_put(atomic_int_t *a, int x)
  __attribute(( PUT(a, x) IN(x) ))
{ lock(&a->lock); a->data = x; unlock(&a->lock); }

void atomic_int_get(atomic_int_t *a, int *x)
  __attribute(( GET(a, x) OUT(x) ))
{ lock(&a->lock); *x = a->data; unlock(&a->lock); }
```

Figure 5: Implementation of atomic channels showing their dataflow annotations.

ables instead to achieve parallelism. The problem with using shared variables to communicate between threads is that it is harder to determine the direction of dataflow through shared variables, which makes it harder for programmers to understand and makes dataflow analysis less precise. To address this issue, SoC-C allows programmers to define new channel types to express directional dataflow.

SoC-C provides the usual array of thread primitives (locks, condition variables, etc.) to allow programmers to create their own channel operations. More importantly, SoC-C provides annotations to allow the programmer to specify that a function performs directional communication. Figure 5 contains a simple example: an "atomic channel" which allows one thread to pass data to another thread atomically. The most important aspect of this example is the annotations. The `PUT(a,x)` attribute specifies that the function argument '`a`' is a channel used to communicate between threads and that the function argument '`x`' is the data transferred into the channel. The `PUT` and `GET` attributes provide important information for the decoupling transformation described in Section 4 and the zero-copy transformation described in Section 8.1. `IN` and `OUT` attributes indicate dataflow through arguments in the usual way.

The PUT and GET attributes were originally added to SoC-C to let us quickly prototype new types of channel without the usual effort of having to add new intrinsic functions to tables in the compiler. We have since realized that most inter-thread communication and communication between threads and stream-oriented devices like Analog-to-Digital Convertors (ADCs) is directional and can be modelled as channels using our annotations. In addition to atomic channels, some examples of channels we use are:

Channel interfaces to ADCs and DACs High rate ADCs usually write data continuously into a circular buffer in memory. In addition to the channel and buffer arguments, it takes a size argument indicating how many samples are required.

```
void ADC_get(adc_t *adc, buffer_t *buf, unsigned sz);
```

Although it interacts with hardware, this function has the same semantics as any other "get" function: if the data is not yet available, the thread blocks until the data is available.

Timed channels provide time-indexed access to data. FIFO channels and atomic channels are at opposite ends of the spectrum on how puts and gets are matched: a FIFO channel matches each get with a unique put; while an atomic channel matches gets with the most recent put. Timed channels provide an intermediate semantic: data is timestamped and a get is matched with the put closest to the requested time.

```
void ts_put(tschan_t *c, int timestamp, void* v);
void ts_get(tschan_t *c, int timestamp, void* v);
```

The `ts_get` operation returns the entry with the closest timestamp to the one specified. All `ts_put` operations must use strictly increasing times and all `ts_get` operations must use strictly increasing times. This restriction allows entries to be discarded when they can no longer be accessed. Timed channels allow for more parallelism between threads since, after the first `ts_put` is performed, `ts_get` operations never block because there is always an entry with a closest timestamp. The cost of this increased performance is less precise synchronization between threads than with FIFO channels: applications that use timed channels are unlikely to give deterministic results.

4. DECOUPLING

In Section 2, we observed that manually introducing pipeline parallelism requires a significant restructuring of the program. There are many papers on avoiding this cost by automating the transformation: Smith [10] applies the technique manually to Cray assembly code which they referred to as "decoupling"; and Palacharla and Smith [9] use program slicing to automatically decouple a program into two threads communicating via FIFO channels: one containing load-store operations, the other containing all other operations. Subhlok et al. [11] have proposed syntax extensions for marking the start and end of pipeline stages within a loop body. These tools allow the programmer to identify what code should be in each section and then the compiler inserts FIFO channel operations as required.

SoC-C's approach is similar in that it requires the programmer to indicate the boundaries between threads. It differs in that the programmer indicates the boundaries by inserting the communication between sections and the compiler determines which code must be in each section — the exact opposite of previous work. In practice, the difference in annotations is usually small: we insert similar annotations at similar parts of the program. We believe our emphasis on communication brings an important benefit: *the programmer is able to select an appropriate channel type in order to reduce synchronization between sections* . These decisions necessarily involve the programmer because using any channel other than a FIFO channel can change the meaning of the program. A secondary benefit is that our channel-based decoupling transformation can be applied to code containing complex control flow and is not restricted to being applied to loops — constraints applied by most prior work.

Figure 6 shows the program in Figure 4 rewritten to use our pipeline construct. There are three major differences. (1) It is possible to write the program as a single loop because decoupling can automatically split it into four parallel copies of the loop. (2) It is not necessary to introduce as many intermediate variables (`samples2`, `samples4`, `bits2`) because our transformation performs range-splitting to split any local variable with disjoint live ranges into multiple variables. (3) It is necessary to use an atomic channel to express the direction of dataflow for the `a_timing` variable.

SoC-C uses the syntax

```
pipeline{
  compound_statement
}
```

to indicate that the body of the compound statement is to be transformed into an equivalent set of parallel sections which

```
atomic_int_t a_timing;
atomic_int_init(&a_timing,0);
pipeline {
  complex_t samples1[2048] @ {M0};
  complex_t samples3[2048] @ {M1};
  complex_t samples5[2048] @ {M2};
  bool      bits1[3024]    @ {M2};
  bool      bits3[3024]    @ {M3};
  int8_t    bytes[378]     @ {M3};
  int       t1;
  int       t2 = 0;
  while (1) {
    ADC_get(&adc,&samples1,2048);
    atomic_int_get(&a_timing,&t1);
    AdjustTiming(t1,samples1) @ P0;
    fifo_put(&f1,samples1);
    fifo_get(&f1,samples1);
    memcpy(samples3,samples1,sizeof(samples1)) @ DMA;
    FFT(samples3) @ P1;
    fifo_put(&f2,samples3);
    fifo_get(&f2,samples3);
    memcpy(samples5,samples3,sizeof(samples3)) @ DMA;
    t2 += FindTimeOffset(samples5)@P2;
    atomic_int_put(&a_timing,t2);
    Demodulate(bits1,samples5) @ P2;
    fifo_put(&f3,bits1);
    fifo_get(&f3,bits1);
    memcpy(bits3,bits1,sizeof(bits1)) @ DMA;
    ErrorCorrect(bytes,bits3) @ Viterbi;
  }
}
```

Figure 6: A version of the program in Figure 2 written using the `pipeline` construct.

communicate (only) using the channel operations already present in the program. Since communication lies at the boundaries between threads, the compiler uses a dataflow analysis which "colors in" the code that lies between the boundaries. This analysis identifies the set of operations that are on the "producer" side of a channel and the set of operations on the "consumer" side of a channel. Repeating this for all channels and considering shared variables, the compiler partitions the operations into sets of operations which must be in the same thread. The body of the `pipeline` construct is then transformed into parallel sections replicating control flow and variables as required.

The decoupling algorithm must make two essential decisions: "What variables and operations to replicate?" and "What operations to place in the same thread?".

The task of decoupling is to split the region of code into as many threads as possible, without introducing timing-dependent behaviour, using channels to communicate between threads. Comparing Figure 4 with Figure 6 we observe that the generated threads do not strictly partition the statements in the original code: some variables and operations (principally those used for control) have been *privatized* (i.e., replicated in multiple threads) while others remain *shared*. The choice of what to privatize is an essential part of the transformation: if too much code or data is privatized, the transformed program can run more slowly and use more memory than the original program. While these decisions could be controlled using annotations on every variable and statement, SoC-C applies some simple default rules that give the programmer control without requiring excessive annotation. By default, scalar variables and variables declared inside the `pipeline` annotation may be privatized. Operations other than function calls may be privatized unless they have side-effects or modify a non-duplicable variable.

The other essential decision that the transformation must make is "What operations must be in the same thread as each other?". To avoid introducing timing-dependent behaviour, the compiler applies the following three rules:

1. To preserve data and control dependencies, any dependent operations must be in the same thread as each other unless the dependency is from a 'put' operations to a 'get' operation on the same channel. This special treatment of dependencies on channel operations has the effect of cutting edges in the dataflow graph.
2. To avoid introducing race conditions, any operations which write to a shareable, non-channel variable must be in the same thread as all operations which read from or write to that variable. Channels are excluded because all channel operations are required to atomically modify the channel.
3. To avoid introducing unwanted non-determinism, all puts to a given channel must be in one thread and all gets from a given channel must be in one thread.

Our implementation of decoupling finds the unique, maximal solution in four stages: live range splitting of privatizable variables, dependency analysis, merging, and thread production. To illustrate our method, we consider the following simple example.

```
 1 pipeline{
 2   for(int i=0; i<100; ++i) {
 3     int x = foo();
 4     if (i % 2 != 0) {
 5       fifo_put(&f,x);
 6       fifo_get(&f,&x);
 7       bar(x);
 8     }
 9   }
10 }
```

The *dependency analysis stage* forms a large number of candidate threads by computing a backward data and control slice [16] from each unprivatized operation ignoring data dependencies on channel operations but including all other operations in the slice. That is, we repeatedly apply rules (1–3) to form candidate threads. In our running example, there are four candidates: one each for `foo()`, `fifo_put(&f,x)`, `fifo_get(&f,&x)` and `bar(x)`.

For example, the candidate for `foo()` is:

```
 2   for(int i=0; i<100; ++i) {
 3     int x = foo();
 9   }
```

the candidate for `fifo_put(&f,x)` is:

```
 2   for(int i=0; i<100; ++i) {
 3     int x = foo();
 4     if (i % 2 != 0) {
 5       fifo_put(&f,x);
 8     }
 9   }
```

and the candidate for `bar(x)` is:

```
 2   for(int i=0; i<100; ++i) {
 3     int x;
 4     if (i % 2 != 0) {
 6       fifo_get(&f,&x);
 7       bar(x);
 8     }
 9   }
```

The *merging stage* combines candidate threads by merging threads that contain the same un-privatizable operation or variable. For example, the candidate for `foo()` is merged

with the candidate for the operation `fifo_put(&f,x)` because they both contain the operation `foo()`. This results in the candidate thread:

```
2   for(int i=0; i<100; ++i) {
3     int x = foo();
4     if (i % 2 != 0) {
5       fifo_put(&f,x);
8     }
9   }
```

This is identical to the candidate for `fifo_put(&f,x)` because the candidate already contained the `x=foo()` operation. Similarily, the result of merging the candidate for `bar(x)` with the candidate for the operation `fifo_get(&f,&x)` is identical to the candidate for `bar(x)`.

The *thread production* stage converts candidate threads to threads by privatizing variables and combining the result using parallel sections.

Syntactic sugar We have found that it is common for pairs of put and get operations to be adjacent. In recognition of this, we added a small piece of syntactic sugar: "`FIFO(x);`". This is equivalent to a put followed by a get on variable `x` and that also declares and initializes a fifo channel. This syntax is illustrated in Figure 7.

5. COMPILER-SUPPORTED COHERENCY

In Section 2, we saw that distributed memory leads to variable fragmentation: if functions running on different processors access the same variable, we must create a version of the variable for each memory region. This is tedious and error prone because it is hard to understand the original design intent. To address this problem, we extend the data placement notation to allow the programmer to express the fact that the additional variables are just versions of the same variable. This preserves the structure of the original design and allows the compiler to detect errors using a single compile-time coherence protocol.

We allow the programmer to assign a variable to multiple memory regions. For example, the declaration

```
bool bits[2048] @ {M2,M3};
```

introduces two copies of the variable `bits`: one in memory `M2` (written `bits@M2`) and one in memory `M3` (written `bits@M3`).

Semantically, the different versions of a variable behave like copies in a coherent cache: an assignment to one version of `bits` (logically) invalidates the contents of the other version. An invalid version of a variable can be made valid by synchronizing it with a valid version of the same variable. The statement

```
SYNC(bits,M3,M2) @ DMA;
```

makes `bits@M2` valid if `bits@M2` was already valid and is an error if `bits@M0` was invalid. A `SYNC` statement is compiled into a copy operation which, in this example, is to be performed on engine `DMA`. Figure 7 illustrates how using variable coherency annotations simplifies the task of keeping track of all the different variables in Figure 6.

Adding support for multiple coherent versions of a variable required the following compiler changes. Various trivial changes to support the new syntax; to transform uses of variables to use the appropriate version; and to transform `SYNC` constructs into `memcpy` operations. Checking for coherence errors is performed using a flow-sensitive, context-insensitive, field-insensitive forward dataflow analysis:

```
atomic_int_t a_timing;
atomic_int_init(&a_timing,0);
pipeline {
  complex_t samples[2048] @ {M0,M1,M2};
  bool      bits[3024]    @ {M2,M3};
  int8_t    bytes[378]    @ {M3};
  int       t1;
  int       t2 = 0;
  while (1) {
    ADC_get(&adc,&samples@M0,2048);
    atomic_int_get(&a_timing,&t1);
    AdjustTiming(t1,samples@M0) @ P0;
    FIFO(samples@M0);
    SYNC(samples,M1,M0) @ DMA;
    FFT(samples@M1) @ P1;
    FIFO(samples@M1);
    SYNC(samples,M2,M1) @ DMA;
    t2 += FindTimeOffset(samples@M2)@P2;
    atomic_int_put(&a_timing,t2);
    Demodulate(bits@M2,samples@M2) @ P2;
    FIFO(bits@M2);
    SYNC(bits,M3,M2) @ DMA;
    ErrorCorrect(bytes@M3,bits@M3) @ Viterbi;
  }
}
```

Figure 7: The effect of rewriting Figure 6 using variable coherency annotations. Changes are highlighted in bold.

1. Each version of each variable can be valid or invalid;
2. A kill makes all versions of a variable invalid;
3. A def to a version of a variable makes that version valid and all others invalid;
4. A `SYNC` statement copies validity from one version of a variable to another;
5. A use checks that the version used is valid; and
6. When flow merges, a version is valid only if it is valid in all incoming edges.

To illustrate the kind of errors these checks detect, suppose the programmer had accidentally reversed the first two arguments in the first call to `memcpy` in Figure 6. Since the programmer's intent is not clear, it would be hard for a compiler to detect this error. In contrast, reversing the `M0` and `M1` arguments in the first `SYNC` statement in Figure 7 is detected as a coherence error by our compiler: the `FIFO` on the previous line defines `samples@M0` which invalidates `samples@M1` but the `SYNC` reads from `samples@M1`.

This coherency mechanism meets our primary goal of supporting safe, statically checked use of distributed memory between processors but within a single thread. Inter-thread coherency checking would require dynamic checking of the ownership of a variable and synchronization — which we think is better expressed using channels. A consequence of not supporting inter-thread coherence is that we perform coherence checking and transformation before decoupling to eliminate coherence annotations before creating new threads.

6. PLACEMENT INFERENCE

Supporting multiple coherent versions of a variable helps communicate the intent of the programmer and, hence, detect errors but it requires that every single use of a variable is annotated. To reduce this annotation burden, we replace *coherence checking* with *placement inference* which exploits the observation that the annotations contain a high degree of redundancy:

```
atomic_int_t a_timing;
atomic_int_init(&a_timing,0);
pipeline {
  complex_t samples[2048];
  bool      bits[3024];
  int8_t    bytes[378];
  int       t1;
  int       t2 = 0;
  while (1) {
    ADC_get(&adc,&samples,2048);
    atomic_int_get(&a_timing,&t1);
    AdjustTiming(t1,samples) @ P0;
    FIFO(samples);
    SYNC(samples) @ DMA;
    FFT(samples) @ P1;
    FIFO(samples);
    SYNC(samples) @ DMA;
    t2 += FindTimeOffset(samples) @ P2;
    atomic_int_put(&a_timing,t2);
    Demodulate(bits,samples) @ P2;
    FIFO(bits);
    SYNC(bits) @ DMA;
    ErrorCorrect(bytes,bits) @ Viterbi;
  }
}
```

Figure 8: An OFDM radio receiver mapped onto a complex architecture using the full set of SoC-C annotations.

- If `P` can only access one memory `M`, and the program contains an RPC "`foo(x)@P`," then variable `x` must be placed in memory `M` and that `x` must have a version in memory `M`.
- If there is only one valid version `x@M` of a variable at the site of a `SYNC(x)` statement, then the only legal source of the SYNC is `x@M`.
- If `x@M` is the only version of a variable whose use is reachable from a `SYNC(x)` statement, then the only sensible target of the SYNC is `x@M`.

These three observations follow a common pattern: if there is only one valid choice, assume that choice is true. Our coherence inference algorithm is similar to flow-sensitive type inference: it uses annotations and the memory topology to add constraints to the system (e.g., an RPC '`f(x)@P`' provides the constraint that '`x`' must be in a memory accessible by 'P' while a reference to '`x@M0`' provides the constraint that '`x@M0`' is valid. Having gathered all the constraints, we use forward-chaining inference to add additional constraints. When no more constraints can be inferred, we choose an open question and test all possible answers to see if they break the constraints. If precisely one possible solution does not break the constraints, then we assume that it is the correct solution and repeat the inference process. This is repeated until no more open questions can be resolved in this way. This process results in a unique solution if there is one because it makes an assumption only if it can prove that all other alternatives are invalid.

In practice we find that SoCs which have multiple memory regions also have sufficiently restricted memory topologies that the compiler can infer most annotations. For example, Figure 8 shows the effect of applying our annotations to the code in Figure 7: all data placement annotations can be inferred in this example.

7. EVALUATING SOC-C ANNOTATIONS

Having completed our presentation of SoC-C, we consider how effective the annotations are at expressing SoC programs. Comparing Figure 8 with the sequential code, we see that to map and parallelize we added: 8 code placement annotations; 0 data placement annotations; 3 SYNC statements; 3 FIFO statements to pass data between threads; and 2 put/get operations on atomic operations.

While annotations and additional statements have been inserted, the structure of the code is unchanged; to port the parallelized code to a new platform, the worst case is that one would delete all the annotations and start again.

Most importantly, we claim that there is little redundancy in the code: most of the changes are independent of the other changes. This suggests that SoC-C allows programmers to express design decisions rather than focussing on the mechanics of making the program correct and consistent.

8. KEY OPTIMIZATIONS

In Section 2 we identified two potential performance issues in our choice of abstraction: the cost of copying buffers into and out of channels; and the cost of using synchronous RPC and threads. This Section describes the (previously known) optimizations we apply to make the cost of these abstractions acceptable.

8.1 Optimizing channels

The semantics of channels is that `put` operations transfer data into the channel and `get` operations transfer data out. This simple semantics ensures that each thread has exclusive access to the data but a literal implementation would require a lot of unnecessary data copying. Network stacks, filesystems and embedded systems often provide a "zero copy" interface which pass pointers instead of copying data. For example the Task Transaction Interface [14, section 4.1.5] splits "put" operations into two operations. "`acquireRoom`" allocates the next empty buffer in a channel; the producer should then write data into the buffer and call "`releaseData`" to make the data available to the consumer. Similarily, "get" operations are split into "`acquireData`" and "`releaseRoom`".

Supporting this style of channel interface required three changes. First, for all channels which hold large buffers, we rewrote the channel implementation to support a zero-copy interface. For example, the zero copy operations corresponding to `fifo_put` are:

```
void fifo_acquireRoom(fifo_t *f, void* *room);
void fifo_releaseData(fifo_t *f, void* data);
```

Secondly, we added attributes to the "put" and "get" functions identifying that these functions could be zero-copy optimized and naming the two associated functions. The augmented set of attributes on the `fifo_put` function is:

```
void fifo_put(fifo_t *fifo, void *data)
  __attribute(( PUT(fifo, data) IN(data)
                ZEROCOPY(fifo1_acquireRoom,
                         fifo1_releaseData) ));
```

Finally, we modified the compiler to analyze the live range of buffers passed to functions with `ZEROCOPY` attributes and insert calls to the two functions at the starts of the live range and at the ends of the live range.

The optimization cannot be used if the live range does not end at a `put` (or start with a `get`), for example, if a variable is put into multiple channels or is used after the `put` operation. For bulk data types, the cost of not optimizing away the copy may be significant so, when zero-copying cannot be used, our compiler reports a warning prompting the programmer to change their code.

```
section{
  complex_t *p_samples2;
  complex_t *p_samples3;
  while (1) {
    fifo_acquireData(&f1,&p_samples2);
    fifo_acquireRoom(&f2,&p_samples3);
    memcpy(p_samples3,p_samples2,...) @ DMA;
    fifo_releaseRoom(&f1,p_samples2);
    FFT(p_samples3) @ P1;
    fifo_releaseData(&f2,p_samples3);
  }
}
```

Figure 9: The effect of zero-copy optimization.

Figure 9 shows the effect of applying zero-copy optimization to the second code section in Figure 4. Using this transformation typically reduces the channel operations to just a small amount of book-keeping. For example, the `put` operation on an atomic channel is split into an operation to wait for the buffer to become available followed later by an operation to release the buffer to other users of the channel. These operations are exactly the same as the lock/unlock operations on a mutex that would normally have been used: our optimizations result in the same low-level, efficient implementation that embedded programmers normally use without the semantic complexity of using shared variables directly.

8.2 Optimizing thread implementation

SoC-C provides synchronous RPCs and uses threads to express sequencing of operations and parallelism. In embedded systems, it is more usual to provide asynchronous (aka non-blocking) RPCs and use an event-driven programming style to express sequencing of operations and parallelism. SoC-C's choice is simpler to use but a conventional thread implementation would incur a large space overhead to store thread contexts and a large time overhead performing context switches when an engine sends an interrupt to signal that it has completed a function call.

To achieve the simplicity of threads with the efficiency of event-driven programming, we use an old trick: we transform threads into event-driven programs [6]. Our compiler transforms each thread into a state machine where states represent points where the program may block on an event and edges are labelled with event handlers which execute code from the thread and update the current state. For example, Figure 10 shows the state machine corresponding to the code in Figure 9.

A typical execution sequence is as follows. The processor spends most of its time in a low-power state waiting for an interrupt with all threads either blocked on a condition variable waiting for a processing element to complete execution or blocked on a channel. On receiving an interrupt signalling completion of a task, the processor invokes an interrupt hander which acknowledges the interrupt and invokes an event handler for the thread currently waiting for that task to complete. This handler typically starts a new task on a processing element and blocks waiting for the task to complete or waiting for the processing element to become available. If the event handler released a lock or put data into a channel before it blocked, the handler may trigger other event handlers: the completion of a single processing element can cause a cascade of event handlers as results propagate to other threads and buffers are freed. When all event handlers in this cascade have executed, all threads are

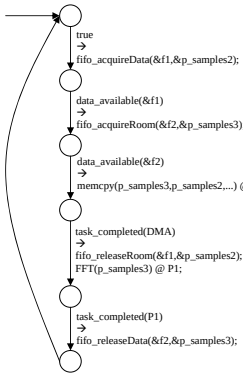

Figure 10: The state machine for Figure 9.

once more blocked on a condition variable or a channel and the control processor returns from the interrupt handler and reenters a low-power state.

8.3 Dataflow Analysis and Phase Ordering

Our compiler relies on the ability to perform a sufficiently accurate dataflow analysis. Since we wish to keep the programmer "in the loop", we limited ourselves to a simple dataflow analysis that was easy to understand. Accordingly, our analysis is flow sensitive, field-insensitive, context-insensitive. Our pointer analysis is just sufficient to analyze programs that use pointers to pass arguments by reference; programmers are encouraged to create abstract data types to hide any other use.

The dataflow analysis used for decoupling and other transformations requires programmer annotations on function arguments. We rely on programmer annotation to determine whether a pointer argument is an 'in' argument (indicated by C's `const` qualifier), an 'out' argument (indicated by an attribute) or an 'in-out' argument (the default). Similarily, if a function accesses a global variable, the function prototype must be annotated to indicate whether it is an 'in', 'out' or 'in-out' variable. If a function modifies just one field in a struct or array, the argument is recorded as an 'in-out' argument indicating that the function does not "kill" the entire argument. In practice, we find that most of this information is already documented informally or can be obtained as a side-effect of function compilation.

Our compiler performs the transformations described in this paper in the following order: dataflow analysis, placement inference/checking, splitting variables with multiple placements into separate variables, zero-copy optimization, decoupling and transforming threads into state machines. Performing dataflow analysis early is important because it allows us to analyze the code before additional pointers are introduced and to give accurate programmer feedback. Coherency checking is performed before decoupling because coherency checking can only be applied within a thread. Zero-copy optimization can be performed either before or after decoupling; to allow it to be performed before decoupling, the '`releaseData`' and '`acquireData`' operations need to be annotated with '`PUT`' and '`GET`' attributes.

Our SoC-C compiler is written as a source to source compiler implemented using Necula et al.'s wonderful CIL [7] C processing framework, 5800 lines of O'Caml code and around 5000 lines of runtime support code including device drivers.

activity	cycles
enter irq handler	10
clearing interrupts	20
start data engine	39
lock overhead	34-38
FIFO transfer overhead	54-55

Figure 11: Performance of SoC-C Implementation

9. PERFORMANCE EVALUATION

This Section evaluates the performance of SoC-C using two criteria: we establish the efficiency of our implementation using microbenchmarks; and we measure how performance of a high performance "software defined radio" application scales with the number of processors.

All measurements were based on a multiprocessor system being developed by ARM Ltd. to implement the physical layer processing of 3.9G mobile phones. This platform centres around a configurable number of moderate-frequency, highly parallel C-programmable data processing engines implemented using ARM's OptimoDE design technology. These processors exploit both data-parallelism using a very wide SIMD (512-bit) datapath and and exploit instruction-level parallelism using VLIW instruction decoding. These OptimoDE engines have a 512-bit data bus to memory and are supported by a DMA engine capable of 512-bit wide transfers. We evaluated using a platform configured to use between 1 and 4 of these data engines as shown in Figure 3. SoC-C code runs on a Cortex-M3 RISC processor with a 32-bit tightly coupled memory. The primary task of the RISC processor is to control data engines, DMA, etc. and to interact with processors executing the higher layers of the network protocol stack. All measurements were made using cycle-accurate models of the data engines, DMA engine, RISC processor and memory system and cycle-approximate models of the peripherals.

9.1 Performance of the runtime system

One of the most important metrics is how long a data engine spends idle between tasks. The control processor must perform the following steps: 1) Complete the current instruction and enter the interrupt handler; 2) Acknowledge the interrupt to the device; 3) Execute the appropriate event handler including constructing a call frame and starting the data engine. Using the simulator we monitored the start/stop signals from data engines, the interrupt signals and the program counter on the control processor and obtained very precise, repeatable measurements to be made (Figure 11). The total time that a data engine is idle between tasks is 69 cycles.

In practice, it is usually necessary to use locks to prevent two threads from using the same engine at once. Locking increases the idle time by 50% to 103–107 cycles. When two threads communicate via a FIFO queue, the time between the completion of a task on one thread and the start of a task in the other thread is 157–162 cycles.

In comparision, our experience is that commercial RTOSs require more than 300 cycles to enter an interrupt handler and trigger a thread context switch. The extra 150-200 cycles may appear negligible until one considers that in that time, our SIMD data engine could have performed another 4500–6000 fixed point multiply operations.

cores	ideal time	actual time	utilization	speedup
1	29286	31101	94%	1.00
2	15013	16865	89%	1.84
4	7876	9077	87%	3.43

Figure 12: Scaling of DVB application.

9.2 Scalability

This Section evaluates how well performance scales as the number of processors is varied using the inner receiver of a Digital Video Broadcast (DVB) physical layer as a benchmark. This has a similar structure and dataflow to our running example but, in addition, it performs: coarse-timing correction to maintain synchronization over long time periods, demultiplexing of data, control bits and pilot channels; channel equalization to correct for fading of individual frequency channels; de-interleaving of the data to reduce sensitivity to bursts of noise. Odd and even symbols require slightly different processing requiring the compiler to decouple code containing if-statements and the two paths have slightly different execution times. Our receiver consists of around 9000 lines of C code split into 17 DSP functions which execute on the SIMD data engines. The total number of cycles of the functions and three DMA transfers is 29286 cycles of which 740 cycles are DMA transfer. Task granularity varies considerably: there are 2 tasks of almost 7000 cycles, 3 tasks of more than 3000 cycles, 1 task of 1000 cycles and the remainder are 500 cycles or less.

We used SoC-C to combine these functions into a single-threaded application and created two pipelined versions of the program for platforms with two and with four SIMD data engines by inserting FIFOs and atomic channels and changing the placement annotations.

We measured the maximum sustainable rate at which a stream of 2K point DVB symbols can be processed measured in cycles per symbol and calculated the best possible time for a system with one DMA engine and N cores given our code placement decisions and function runtimes and ignoring data dependencies which would prevent perfect parallelization. Ignoring data dependencies makes this number a little conservative (too low). We calculated utilization as the ratio between the ideal rate and the actual rate and calculated speedup as the ratio of actual rate against the actual rate of the 1-core variant. The results are summarized in Figure 12. The results for a single core demonstrate the effectiveness of our implementation strategy: the overhead of using SoC-C is just 1800 cycles (6%) which matches our expectation from the microbenchmarks. On two cores, the application speeds up by a factor of 1.84 compared with the single core version. We were unable to achieve perfect speedup because the coarse granularity of tasks made it impossible to perfectly balance the load. On four cores, the application speeds up by a factor of 3.43 compared with the single core version despite coarse task granularity.

10. DISCUSSION AND RELATED WORK

SoC-C's major influences are stream programming languages such as StreamIt [5] which emphasize pipeline parallelism and have a clear separation of the communication language from the kernel language. We maintain the separation of communication/control layer (SoC-C) from computation (code called by RPCs) but we chose a sequential communication language instead of a dataflow language because we found

it hard to express global control (i.e., conditionals that span multiple pipeline stages) over pipeline stages that execute asynchronously with respect to each other. Using decoupling to introduce parallelism, gives the ease of expression of global control that imperative languages provide combined with the pipeline parallelism that stream languages provide.

Decoupling has been applied many times; we cite a representative sample. Smith [10] applies the technique manually to Cray assembly code to separate load-store operations from other operations to program Access-Execute processors; [9] automated this transformation; [4, 8, 2, 3] decouple programs automatically based on load-balancing heuristics; [11, 13] rely on programmer annotations to mark the beginning and end of pipeline stages. All these papers rely on partitioning of the operations into pipeline stages and then inserting FIFO channels. Our channel-based decoupling algorithm does the opposite: it relies on the programmer inserting channels and partitions the operations accordingly. The difference is small but significant: making the channels first-class concepts, instead of mere implementation details, lets the programmer use different channel types to explicitly relax synchronization between pipeline stages to avoid loss of decoupling. We are not aware of any work that uses non-FIFO channels when automatically decoupling a sequential program though StreamIt's "teleport messaging" [12] provides a related feature for dataflow languages.

Although SoC-C borrows syntax from OpenMP, the two languages target very different systems and parallelism patterns: OpenMP targets SMP systems and supports data parallelism using annotations on for-loops; SoC-C targets AMP systems and, hence, supports pipeline parallelism.

EXOCHI [15] also tackles the problem of programming heterogeneous multicore systems but is complementary since they focus on coping with multiple instruction sets/toolchains, providing shared virtual memory and dynamically allocating tasks to accelerators whereas we focus on distributed memory, static allocation of tasks and decoupling.

There has been a large body of work on software distributed shared memory and on reducing cache-coherency traffic between threads using compiler techniques. SoC-C's approach is to express inter-thread communication (which requires dynamic checks) using channels and restrict coherence checking for intra-thread, inter-processor communication (which our compiler checks statically).

SoC-C handles data copying differently from many systems: RPCs normally copy bulk data structures; FIFO channels normally copy data both on a put and a get; private memories are often used to store local copies of variables whose master copy is in shared memory. Instead, SoC-C gives explicit control over data copying and SoC-C provides support to make this less burdensome and error-prone.

11. CONCLUSIONS

Mapping an application onto low-power, high-performance SoCs is a challenging problem due to the architectural complexity needed to achieve high energy efficiency. A common approach to the problem of complex hardware is to use software libraries to hide the complexity from the user. To achieve significantly higher energy efficiency we take a different approach: SoC-C provides the programmer with explicit control over how an application is mapped onto an architecture without requiring significant manual restructuring. Any language requires careful implementation and choice of optimizations to minimize overhead: our compiler is able to speedup a coarse-grained, real-world application by a factor of 3.4 on a four-core platform achieving utilization of 87%.

12. REFERENCES

[1] P. L. Bird, A. Rawsthorne, and N. P. Topham. The effectiveness of decoupling. In *International Conference on Supercomputing*, pages 47–56, 1993.

[2] M. Bridges et al. Revisiting the sequential programming model for multi-core. In *MICRO 2007: Proc. of Symposium on Microarchitecture*, 2007.

[3] J. Dai, B. Huang, L. Li, and L. Harrison. Automatically partitioning packet processing applications for pipelined architectures. In *PLDI 2005*, pages 237–248, 2005.

[4] W. Du, R. Ferreira, and G. Agrawal. Compiler support for exploiting coarse-grained pipelined parallelism. In *Proc. of Conf. on High Performance Networking and Computing (SC2003)*, page 8, 2003.

[5] M. I. Gordon et al. A stream compiler for communication-exposed architectures. In *Proc. Architectural Support for Programming Languages and Operating Systems*, pages 291–303, 2002.

[6] H. Lauer and R. Needham. On the duality of operating system structures. In *Proc. Symposium on Operating Systems*, 1978.

[7] G. C. Necula et al. CIL: Intermediate language and tools for analysis and transformation of C programs. In *CC '02: Proc. Int. Conf. on Compiler Construction*, pages 213–228. Springer-Verlag, 2002.

[8] G. Ottoni, R. Rangan, A. Stoler, and D. I. August. Automatic thread extraction with decoupled software pipelining. In *MICRO '05: Proc. Int. Symposium on Microarchitecture*, Nov 2005.

[9] S. Palacharla and J. E. Smith. Decoupling integer execution in superscalar processors. In *MICRO 28: Proc. of International Symposium on Microarchitecture*, pages 285–290, 1995.

[10] J. E. Smith. Decoupled access/execute computer architectures. *ACM Trans. Comput. Syst.*, 2(4):289–308, 1984.

[11] J. Subhlok et al. Exploiting task and data parallelism on a multicomputer. In *Proc. of Symp. on Principles and Practice of Parallel Programming*, 1993.

[12] W. Thies et al. Teleport messaging for distributed stream programs. In *PPoPP '05: Proc. of Symposium on Principles and Practice of Parallel Programming*, pages 224–235. ACM Press, 2005.

[13] W. Thies et al. A practical approach to exploiting coarse-grained pipeline parallelism in C programs. In *MICRO 2007*, 2007.

[14] P. van der Wolf et al. Design and programming of embedded multiprocessors: An interface-centric approach. In *CODES+ISSS'04: Hardware/Software Codesign and System Synthesis*, 2004.

[15] P. H. Wang et al. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system. In *Proc. PLDI*, 2007.

[16] M. Weiser. Program slicing. In *ICSE '81: Proc. of International Conference on Software Engineering*, pages 439–449, 1981.