

Fast Paths in Concurrent Programs

Wen Xu

Department of Computer Science
Princeton University

wxu@cs.princeton.edu

Sanjeev Kumar

Intel Labs
Intel Corporation

sanjeev.kumar@intel.com

Kai Li

Department of Computer Science
Princeton University

li@cs.princeton.edu

ABSTRACT

Compiling concurrent programs to run on a sequential processor presents a difficult tradeoff between execution time and size of generated code. On one hand, the process-based approach to compilation generates reasonable sized code but incurs significant execution overhead due to concurrency. On the other hand, the automata-based approach incurs a much smaller execution overhead but can result in code that is several orders of magnitude larger.

This paper proposes a way of combining the two approaches so that the performance of the automata-based approach can be achieved without suffering the code size increase due to it. The key insight is that the best of the two approaches can be achieved by using symbolic execution (similar to the automata-based approach) to generate code for the commonly executed paths (referred to as fast paths) and using the process-based approach to generate code for the rest of the program. We demonstrate the effectiveness of this approach by implementing our techniques in the ESP compiler and applying them to a set of filter programs and to VMMC network firmware.

1. INTRODUCTION

Concurrency is a convenient way of structuring programs [19, 24] in a variety of domains including embedded devices [6, 17], user interfaces [10, 31], programmable devices [21], servers [32], media-processing applications [20], and network software stack [4, 29]. This is often true even when these programs are written to run on a single processor. This is because programs in these domains are required to simultaneously process multiple external events at the same time. Concurrent programs have multiple threads of control that coordinate with each other to perform a single task. The multiple threads of control provide a convenient way of keeping track of multiple contexts in the program.

Figure 1 shows a concurrent program with 5 concurrent threads of control (P1-P5). On a uniprocessor, the entire concurrent program needs to be compiled to run efficiently on a single processor. On a multiprocessor (with 2 processors), the program is partitioned so that P1-P3 are run on the first processor while the remaining program is run on the second processor. In each case, several threads of control have to be compiled to efficiently run on a sequential processor. This paper addresses this problem.

The overhead of implementing concurrency on sequential processors can be substantial. There are two main approaches to compiling concurrent programs on sequential processors: process-based approach and automata-based ap-

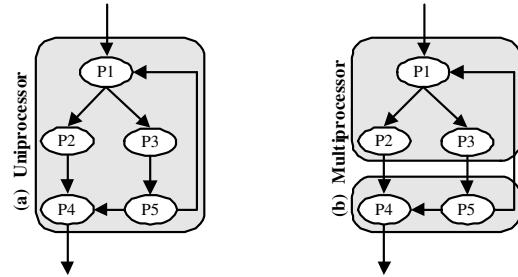


Figure 1: Concurrent Program

proach (Section 4). The two approaches differ radically in the concurrency overhead and the size of the generated code. A study [15] evaluated the two approaches on a set of concurrent programs written in Esterel [6]. The study found that the automata-based approach resulted in code that was twice as fast as the process-based approach. However, the size of the code generated by the automata-based approach was 2-3 orders of magnitude larger than that produced by the process-based approach.

This paper proposes a technique for extracting and optimizing fast paths from concurrent programs so that the performance benefit of the automata-based approach can be achieved without significantly increasing the size of the generated code. A *fast path* is a commonly executed path in a program. Substantial performance improvements can be achieved by aggressively optimizing the fast paths in the programs. Past research [30, 26, 23] has focused on fast paths in sequential programs.

In the absence of automatic fast path extraction, fast paths are often implemented manually by the programmer [21]. The programmer can insert a predicate in the program to check for the common case and transfer control to the fast path code. The fast path code has to be functionally equivalent to the corresponding path in the program. The programmer is responsible for manually extracting and optimizing the fast path code.

Although, manually extracting fast paths often results in good performance, it suffers from three drawbacks. First, manually extracting fast paths involves substantial programmer effort. Second, manually implementing fast paths is very error prone. Fast paths violate abstraction boundaries and rely on global information like the state of the different threads of control and their local data structures. As the program evolves, for each change made to the concurrent program, a corresponding change has to be made to the fast

path. In addition, programmers often introduce subtle bugs when they try to aggressively optimize fast paths. Third, it is difficult to build robust fast paths. One often ends up with fast paths that help simple applications with very predictable process interactions but have little impact on applications. The more specific the fast path predicate (that identifies the fast path), the more specialized and efficient fast path will be. However, it also means that the predicate will be satisfied less often. It is easier to experiment with various fast paths and identify the right one to employ if the programmer effort needed to build a fast path is small.

Summary of Contributions. This paper presents techniques to automatically generate fast paths in concurrent programs using a compiler. This approach avoids the drawbacks associated with manual fast path extraction while providing the performance benefits. The main contributions of this paper are the following:

1. It extends the traditional definition of fast paths to make them more flexible (Section 2.2).
2. It proposes a variant of path expressions that not only allows the programmer to specify fast paths in concurrent programs but also lets them specify the scheduling choices made on the fast path. A key feature of the fast path specifications is that they are just hints—while they improve the performance of the program, they do not change the semantics of the program and, therefore, do not affect program correctness. (Section 3)
3. It presents a technique for extracting and optimizing fast paths in concurrent programs. This technique delivers the performance of the automata-based approach without the associated blowup in the size of the generated program. One important aspect of this technique is that it preserves the fairness guarantees of the program. (Section 4)
4. It provides a practical demonstration of the approach by implementing the techniques described in the context of the ESP [21] compiler. A set of filter programs and VMMC network firmware are used to evaluate the effectiveness of the technique. On filter programs, our technique outperforms even the automata-based approach without any dramatic increase in the size of the generated code. On VMMC firmware, our technique achieves up to 22% improvement in latency and up to 40% improvement in bandwidth over the process-based approach. (Section 6)

2. PROBLEM STATEMENT

This paper presents a technique to reduce the concurrency overhead in concurrent programs by extracting and aggressively optimizing fast path code. This section starts with a description of a simple concurrent language and an example that is used throughout this paper. It then describes fast paths in detail. Finally, it discusses the scope of the paper.

2.1 Concurrent Programs

In this section, we describe a simple concurrent programming language that we use to demonstrate the techniques presented in this paper.

In this language, concurrency is expressed using processes and channels. A program consists of a set of processes communicating with each other over channels. Each process

```

1: channel hostSendRequestC( int, int ) external out;
2: channel hostFetchRequestC( int ) external out;
3: channel translateRequestC( int, int );
4: channel translateReplyC( int, int );
5: channel dataSendC( int, int );
6: channel networkSendC( int ) external in;
7:
8: process hostRequest {
9:   var virtualAddress, physicalAddress, size, source;
10:  while (true) {
11:    alt {
12:      in( hostSendRequestC, virtualAddress, size) {
13:        out( translateRequestC, virtualAddress, size);
14:        while ( true) {
15:          in( translateReplyC, physicalAddress, size);
16:          if ( size == 0) break;
17:          out( dataSendC, physicalAddress, size);
18:        } // while
19:        #1
20:      } // in
21:      in( hostFetchRequestC, source) {
22:        // Code omitted
23:      } // in
24:    } // alt
25:  } // while
26: }
27:
28: process translateAddress {
29:   constant pageSize = 4096;
30:   var virtualAddress, physicalAddress, size;
31:   while (true) {
32:     in( translateRequestC, virtualAddress, size);
33:     assert( virtualAddress % pageSize == 0);
34:     assert( size % pageSize == 0);
35:     while ( size > 0) {
36:       if ( translationUnavailable(virtualAddress)) { #2
37:         // Code to fetch the translation entry
38:       } // if
39:       physicalAddress = translate( virtualAddress);
40:       out( translateReplyC, physicalAddress, pageSize);
41:       size = size - pageSize;
42:     } // while
43:     out( translateReplyC, 0, 0); // Done
44:   } // while
45: }
46:
47: process networkSend {
48:   var physicalAddress, size, packet;
49:   while (true) {
50:     in( dataSendC, physicalAddress, size);
51:     packet = preparePacket( physicalAddress, size);
52:     out( networkSendC, packet); #3
53:   } // while
54: }

```

Note: The `assert` statements in the code are used to state the assumptions made to simplify the example. The `#1` and `#2` are annotations to mark statements and are used in Section 3.

Figure 2: A Running Example. (Illustrated in Figure 3)

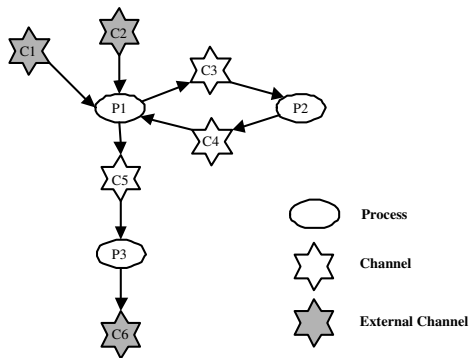


Figure 3: Example. Illustration of the example in Figure 2. Process P1 is `hostRequest`, P2 is `translateAddress`, and P3 is `networkSend`. Channel C1 is `hostSendRequestC`, C2 is `hostFetchRequestC`, C3 is `translateRequestC`, C4 is `translateReplyC`, C5 is `dataSendC`, and C6 is `networkSendC`.

represents a sequential flow of control in the concurrent program.

Processes communicate with each other over channels. Messages are sent over the channels using the `out` operation and received using the `in` operation. Communication over channels is synchronous¹ or unbuffered—a process has to be attempting to perform an `out` operation on a channel concurrently with another process attempting to perform an `in` operation on that channel before the message can be successfully transferred over the channel. Consequently, both `in` and `out` are blocking operations. The `alt` statement allows a process to wait on `in` and `out` operations on several different channels till one of them becomes ready to complete.

External channels allow the concurrent program to communicate with external world (for instance, to send a packet into the network). External channels are like regular channels; the only difference is that they have an external reader or a writer.

In the presence of nondeterminism (due to the `alt` statement), the language guarantees *fairness*.² When multiple channel operations are ready in an `alt`, if the implementation always chooses one particular channel operation, the processes waiting on the other channels can be starved out. This is referred to as *unfairness*. Fairness, therefore, implies freedom from starvation. It should be noted that fairness does not imply that each of the enabled guarded statements will be chosen with equal probability.³

In addition to channel operations, the language supports the common control flow statements like `if-then-else` and `while` statements. For simplicity, it supports just one type of data: integers.

¹Also known as *rendezvous* channels.

²Two types of fairness guarantees can be provided: *weak fairness* and *strong fairness* [2]. However, the fast path extraction technique described in this paper preserves the fairness semantics for both these types. Consequently, we do not make a distinction between the two in the rest of this paper.

³The term “fairness” is sometimes used in the operating systems community to imply this meaning. In this paper, we always use the term *fairness* to only imply starvation-freedom.

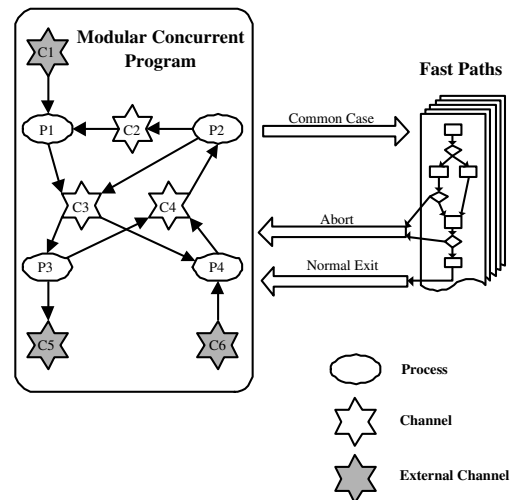


Figure 4: Fast Paths in Concurrent Programs

Figure 2 and Figure 3 show a code fragment that is used as a running example in this paper. It is extracted from our VMMC firmware code [21] for a gigabit network card. The code shows the steps involved in sending a packet in the VMMC firmware [21]. When the user application has some data to send, it sends a request via channel `hostSendRequestC` to process `hostRequest`. After process `hostRequest` gets the request, it first translates the virtual address to physical address, and then sends the data page by page to the destination. Process `hostRequest` consults process `translateAddress` for address translation. Process `translateAddress` has a table which caches recently translated addresses. On a table hit, the physical address is immediately available. Otherwise it needs to fetch corresponding translation. For messages more than one page, process `translateAddress` returns the physical address for each page. Then process `hostRequest` makes a request to process `networkSend` to actually send the page onto the network.

2.2 Fast Paths

A *path* [26, 3] is a dynamic execution path in a program. Typically, a small set of paths in the program account for a large percentage of its execution time.

A *fast path* provides better performance to a set of commonly executing paths in the program. It should be emphasized that a fast path is not necessarily a single execution path in the program. A fast path is typically a set of related execution paths in the program.

Traditionally, a fast path [30, 26, 23] consists of two components: A predicate that identifies a common case, and specialized code that is optimized to efficiently handle that common case. As long as the predicate holds, executing the specialized code is functionally equivalent to the original execution path chosen without the fast path. It is formed by extracting code fragments from several different modules. This allows fast paths to avoid module-crossing overheads; it also makes them more amenable to compiler optimizations.

In this paper, we extend the traditional notion of a fast path to allow them to abort midway through the execution (Figure 4) for two reasons. First, it is often difficult to iso-

late the fast path with a single predicate that has to hold at the start of the fast path. Second, in some cases, a predicate might not hold at the start of the fast path but might become true later. For instance, a DMA engine⁴ might not be available at the start of the fast path but might become available by the time it is needed at a later point on the fast path.

2.3 Scope

To extract fast paths from programs, three questions need to be answered.

1. How are fast paths selected? This requires knowledge about which paths in the program are critical as well as commonly executed.
2. How does the programmer specify the fast paths in the program? The compiler will use this information to aggressively optimize the fast path.
3. How does the compiler extract and optimize the fast path?

In this paper, we address the last two questions: specifying and optimizing fast paths. Identifying fast paths is an independent problem that is not addressed here. In this paper, we assume that the programmer identifies the fast paths either based on knowledge of the application behavior or by using some recent work on path profiling in sequential [3, 22] and parallel programs [11, 31]. The work presented in this paper can also simplify the task of identifying fast paths. This is because a programmer (or even an automated tool) can try out several different fast paths with little effort to determine the most profitable fast path.

3. SPECIFYING FAST PATHS

Traditionally, fast paths in sequential programs are often specified by annotating the program to indicate the “likely” result (**true** or **false**) of conditional statements of the program [27]. The HIPPCO [11] compiler allows a probability to be specified for conditional statements. These probabilities can be determined by program profiling. Another approach [23] is to use a predicate to specify fast paths. The compiler then extracts the fast path code by partially evaluating the code based on the predicate. Neither of these approaches meets our needs.

To specify fast paths in concurrent programs, we identified three desirable properties that the fast path specification mechanism should satisfy. First, the fast path specification should be just hints to the compiler and, therefore, should not affect the correctness of the program. In addition, since they are just hints, the fast path specification should be kept separate from the code to the extent possible. This would ensure that the specifications do not make the code less readable by cluttering it. Second, the fast paths should have the ability to abort prematurely (Section 2.2). Finally, the specification should allow programmer to control the scheduling of the different processes involved in the fast path since it can have a big impact on the performance. Note that the traditional approaches described in the previous paragraph do not satisfy these properties.

This paper proposes using an extension of path expressions [9, 8] to specify fast paths in concurrent programs. A path expression is a regular expression over control points in a program. Path expressions provide a succinct and pow-

⁴A DMA engine allows a device to move bulk data efficiently.

```

fastpath demo {
  process hostRequest {
    statement  hostSendRequestC as H0,
               translateRequestC as H1,
               translateReplyC as H2,
               dataSendC as H3,
               #1;
    start      H0 ? (size < 10000);
    follows    H1 ( H2 H3 )* ;
    exit       #1;
  }
  process translateAddress {
    statement  translateRequestC as T1, #2;
    start      T1 <=> H1;
    exit       T1;
  }
  process networkSend {
    statement  dataSendC, #3;
    start      dataSendC;
    exit       #3;
  }
}

```

Note: ‘#1’, ‘#2’, and ‘#3’ name the first statement after the point where they appear. The ‘?’ is used to specify a predicate that has to hold at the statement. The ‘<=>’ is used to specify the statement in the other process with which it is communicating. The **as** allows the programmer to specify a shorter name for a statement.

Figure 5: A Fast Path Example.

erful way of expressing control flow in programs and have been widely used (Section 7).

We will now illustrate our fast path specification language with a fast path (Figure 5) in our example (Figure 2). Four fields can be specified for each process involved in the fast path. The **statement** field enumerates the list of all statements that are relevant to the fast path. The **start** field specifies the starting *statement element* while the **exit** field specifies the statement element that marks the end of the fast path in that process. The **follows** field is a regular expression on statement elements that specifies the set of execution paths that the process can take between **start** and **exit**. The fast path is terminated if either the **exit** element is satisfied or if it deviates from the path specified by the **follows** field.

Three points are worth noting here. First, the **follows** and **exit** fields are optional. Second, any statement that is not explicitly included in the **statement** field has no impact on whether or not an execution path is selected on the fast path. This helps to keep the regular expression small. For instance, in process **translateAddress**, the fast path is aborted if it encounters statement ‘#2’ (Figure 5). However, statements involving operations on channel **translateReplyC** are simply ignored while determining if a particular path belongs to the fast path because it is not listed in the **statement** field. Third, the **exit** field is redundant in process **networkSend** (Figure 5). This is because it specifies a null path expression for the **follows** field. Consequently, the fast path would terminate if it encountered either of **dataSendC** or ‘#3’ after starting the fast path as it would no longer satisfy the **follows** field.

A *statement element* is the basic unit in the fast path

specification. In the simplest case, a statement element is just a statement.⁵ A statement element can also qualify a statement with one or more of the following. First, a predicate can be specified that has to hold at that statement. For example, the start condition in process `hostRequest` specifies that the predicate `size < 10000` has to hold. Second, a statement involving a channel operation can specify the statement in the other process with which it is communicating (using '`<=>`'). Finally, it can explicitly specify the scheduling decisions on the fast path and override the default scheduling policy.

Our default scheduling policy works as follows: At the start, all processes on the fast path that are ready to run (i.e. unblocked) are placed in a FIFO ready queue (in the order the processes appear in the fast path specification). The execution begins with the first process on the queue and proceeds until a channel operation is encountered. If the channel operation causes it to block, the next process from the ready queue is picked and executed. Alternately, if the currently executing process communicates with a blocked process that is part of the fast path, the process performing the `in` (read) operation is the one that continues while the other process is added to the ready list.

The default scheduling policy works well in practice because it often reflects the critical path in the concurrent program. For instance, the default policy picks the best scheduling for the example in Figure 2 (which was extracted from a real program [21]). In addition, copy propagation optimization is very effective with this policy because the process reading from the channel is likely to use those values immediately.

In a few rare cases, different scheduling decisions (from the ones made by the default scheduling policy) at a few locations on the fast path can improve performance. We provide a simple yet powerful mechanism to override the default scheduling decision. An element can be qualified with a `yield` directive that allows the currently scheduled process to specify a different process to be scheduled immediately. For instance, suppose (`H2 yield translateAddress`) were used in the place of `H2` in the `follows` field of process `hostRequest`. In this case, after the communication on channel `translateReplyC`, process `translateAddress` would be scheduled to run instead of process `hostRequest`.

4. GENERATING FAST PATHS

4.1 Background

This paper focuses on the application domains that use concurrency as a convenient way to structure programs even on a uniprocessor (Section 1). Consequently, the most efficient way to execute these programs is to run all its processes in the same virtual address space (i.e. single operating system process) and perform scheduling at the user level in the runtime system. There are two main approaches to implement this: *process-based* approach and *automata-based* approach.

The *process-based* approach [28, 15, 21] is the popular approach to compile concurrent programs to run on sequential processors. In the process-based approach, the compiler generates the code for each process separately and inserts additional code to periodically context switch between them so that all the processes make forward progress.

The advantage of the process-based approach is that the size of the generated program is reasonably small—It is roughly the sum of the sizes of the individual processes. However, the generated code incurs a runtime overhead due to the concurrency. The runtime overhead stems from three sources. First, a context switch involves saving the state of the running process, and then retrieving the state of the next process and running it. Second, when values are transferred over a channel, there is overhead associated with it that is similar to the overhead of passing parameters to a function. Finally, nondeterministic statements require a mechanism (like randomly picking between the available options) that guarantees fairness.

The *automata-based* approach [10, 6, 12, 29] is a radically different approach that uses *symbolic execution* to generate code for concurrent programs. Symbolic execution is a general technique that has been applied in wide variety of areas including program testing, model checking, program analysis, and optimization. In the automata-based approach, symbolic execution is used to enumerate the control state space of a concurrent program. We explain this briefly in the following paragraph (See [29] for a detailed description).

The automata-based approach essentially treats each process in the concurrent program as a state machine and combines all the state machines in the program to generate a single global state machine. Each statement in a process represents a state in the corresponding state machine. A tuple consisting of the state of each of the various state machines denotes a state of the global state machine. At each step, the global state machine takes a state in one of the individual state machines. This is repeated until all the transitions reachable from the start space are explored. It should be noted that the nondeterminism in various processes of the concurrent program gets translated into nondeterministic transitions in the global state machine. Consequently, the global state machine is essentially a sequential program with nondeterminism.

The advantage of the automata-based approach is that there are no context switches and channel operations in the generated code. Although, there is still overhead involved due to the nondeterminism, the code generated is extremely fast. The disadvantage of this approach is that the global state machine generated can be, in the worst-case, exponential in the size of the individual state machines. Some optimization techniques [12, 11] alleviate the code blowup problem by identifying and eliminating some of the duplicated code. Still, the code blowup remains exponential in the worst-case.

Edwards *et al.* [15] compared the two approaches on a set of Esterel programs. Esterel (Section 7) is a deterministic concurrent language. The study found that the automata-based approach resulted in code that was twice as fast as the process-based approach. However, the size of the code generated by the automata-based approach was 2–3 orders of magnitude larger than that produced by the process-based approach. Such a large increase in the size of generated code is often unacceptable.

⁵Statements in a process are identified either using a channel name (when the channel name uniquely identifies a statement that performing an operation on it) or using the '#' annotation. For instance, the '#2' refers to the first statement in the body of the `if` statement in process `translateAddress`.

4.2 Extracting Fast Paths

This paper proposes combining the two approaches so that the generated code achieves the performance of the automata-based approach while resulting in code size similar to that from the process-based approach. The key insight is that the best of the two approaches can be achieved by using symbolic execution (similar to the automata-based approach) to generate code for the fast paths and using the process-based approach to generate code for the rest of the concurrent code.

Our approach proposes generating code for the concurrent program in three stages:

1. Process-based Baseline Code. The compiler uses process-based approach to generate code for the program. This portion of the code is a complete stand-alone implementation of the program.

2. Extracting Fast Path Code. The compiler uses the fast path specification to generate code for the fast paths. For each fast path specified, the compiler first translates the path expressions (one for each process involved) into finite-state machines. Each state-machine includes a *start* state and a *normal exit* state which corresponds to the start and the end of the fast path. Then, the compiler uses symbolic execution to follow all possible execution paths from the start of the fast path. During the symbolic execution of each execution path, the compiler makes transitions in the corresponding state machines each time an “interesting” location⁶ is encountered in the program. The situation when such a transition is not available corresponds to a path that no longer matches the fast path specification. When this happens, the execution point (where the violation is first detected) is marked as an *abort* point and that execution path is not longer executed symbolically. Similarly, if the state machine reaches the *normal exit* state, the execution point is marked as a *normal exit* point and that execution path is terminated.

Figure 6 illustrates the symbolic execution performed to extract the fast path specified in Figure 5 for the example in Figure 2 & Figure 3. Each state in this figure includes the program counter (line number from Figure 2) for each process in the fast path. The starting state (as specified in Figure 5) corresponds to the state (P1=13, P2=32, P3=50). At each step, one of the processes is being symbolically executed. For example, at state (P1=15, P2=41, P3=50), process P2 is chosen to be executed. It symbolically executes the statement `P2.size=P2.size-pageSize` on line 41 and changes its program counter to 35. Two processes may also communicate. For example, at the start state (P1=13, P2=32, P3=50), process P1 and P2 communicate on channel `translateRequestC`. Therefore the two processes update their program counter and the executions enters the next state (P1=15, P2=35, P3=50). Any state in which one of the processes has reached an end state or deviated from the specified path is marked as an exit state or an abort state respectively. No transition out of such a state is considered. For example, in state (P1=19, P2=32, P3=50), process P1 has reached the exit state, so the symbolic execution does not proceed any further on this path.

⁶Recall that the fast path specification specifies all locations in the program that are relevant to it.

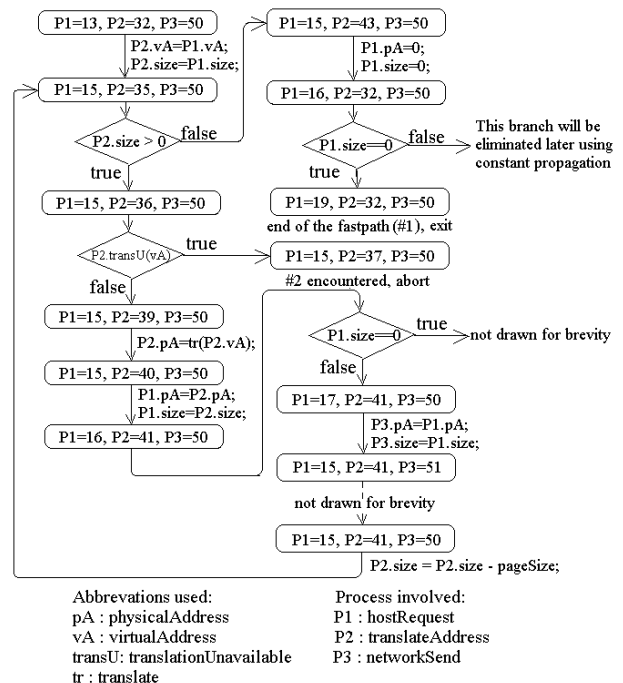


Figure 6: Fast Path Extraction

3. Entering and Exiting Fast Path. The process-based code and the fast path code are then combined by adding code that transfer control to each other (Figure 4). In the process-based code, code is inserted at the appropriate points to check if the starting condition for the fast path is satisfied and, if it is satisfied, transfer control to the fast path code. In the fast path code, code is added at the exit points (normal exit and abort points) that return control to the process-based code.

We need to do two things when transferring control between the process-based code and the fast path code. First, we need to update the “program counter” pointer for each involved process, this pointer identifies which instruction in the process is being executed. Second, each process has a state variable that remembers which channel operations are ready in an `alt`; these state variables need to be updated.

4.3 Process Scheduling on the Fast Path

A fast path usually involves multiple processes. Therefore, process scheduling decisions have to be made during symbolic execution. Since the scheduling decisions have a big impact on the performance, the programmer is allowed to precisely specify the scheduling decisions to be made on the fast path (Section 3).

Our compiler uses the specified scheduling policy during fast path extraction. As the start of the fast path, it puts all the processes involved in the fast path that are ready to be executed (i.e. unblocked) in a *ready list* (FIFO) and starts symbolically executing the first one (say P_A). It follows this process until it encounters a channel operation or a `yield` directive to yield to a different process (say P_B) (Section 3). At this point, there are three possibilities. First, if it (P_A) encounters a channel operation and the channel

operation blocks, the symbolic execution picks up the next process (say P_C) in the ready list and symbolically executes it. Second, if the channel operation can complete and involves communication with another process (say P_D), one of the two processes P_A and P_D is picked to be symbolically executed next (based on the scheduling decision specified) and the other is put in the ready list. Finally, if it (P_A) encounters a yield directive, P_B is extracted from the ready list to be symbolically executed while P_A is put on the ready list.

4.4 Fairness on the Fast Path

The generated code is required to preserve fairness semantics of the program (Section 2). Fast path involves code that is extracted from different processes and thereby makes some scheduling decisions. The compiler has to ensure that it does not introduce starvation in the program. Starvation can arise from two situations. Either the specified fast path can be infinitely long (due to the presence of the repetition operator in the path expressions). This can potentially allow a fast path involving two processes to starve out a third process. Or it arises if the nondeterminism is not handled correctly on the fast path.

Our compiler employs a simple strategy to handle fairness: it simply relies on the underlying process-based code that already handles fairness. To avoid starvation due to the infinitely long fast paths, it places a bound (by maintaining counters for each repetition operator) on how long the fast path can execute and aborts the fast path if the bound is exceeded. Then the control would be returned to the process-based code which ensures fairness. To avoid starvation due to nondeterminism, the generated code periodically chooses not to execute the fast path code even when the starting conditions for that fast path are satisfied. Note that this is required for only those fast paths that include a nondeterministic choice. This means that the process-based code that ensures fairness is executed a fraction of the time even when the starting conditions for that fast path are satisfied. This is sufficient to ensure fairness in the generated code even if the fast path code does not handle nondeterminism fairly.

Our approach to handle fairness on the fast path is not only simple but also presents an opportunity to avoid the overhead due to nondeterminism in the fast path. Recall that the nondeterminism overhead is incurred even by the automata-based approach. Consequentially, fast paths can potentially deliver better performance than the automata-based approach. As explained in the previous paragraph, the fast path code is not required to handle nondeterminism fairly. This allows the fast path to arbitrarily pick a nondeterministic choice while completely ignoring other choices. Therefore, on encountering a nondeterministic statement, the compiler uses the fast path specification to narrow down the choices to only those that stay on the fast path. In this case, the compiler is using the fast path specification to guide the symbolic execution. So far, it had been only used to check if the symbolic execution stayed on the specified path and abort the fast path if the execution strayed from it.

4.5 Optimizations on Fast Path

In addition to the optimization described in Section 4.4, A number of optimizations are applied to improve the per-

formance of fast path code.

Enabling Traditional Optimizations. Traditional optimizations, like copy propagation and dead code elimination, on fast paths result in program specialization and cross-module optimizations. The fast path is composed of fragments of code extracted from several processes. Since the code is executed only when the starting condition is satisfied, the fast path code can be specialized assuming the starting conditions. In addition, since process⁷ boundaries are eliminated while extracting fast paths, the optimizations on fast paths are effectively cross-module optimizations.

However, traditional optimizations cannot be directly applied to the fast paths in isolation. This is because their control flow is linked back to the rest of the code via the abort/exit points. To solve this problem, we need to propagate some information back from each of the individual processes to the fast paths. For instance, we perform live-variable analysis on the fast path in two stages. First, we perform live variable analysis in each of the processes. Second, this liveness information is propagated to each of the abort/exit point in the fast path depending on where the exit/abort point in the fast path returns control to the various processes. Using this information, the liveness analysis can be performed on the fast path.

Speeding up fast path using lazy execution. Some code can be eliminated by rearranging the sequence of execution on the fast path. For example, a lot of assignments are done on the fast path to mimic message passing between processes to update their local variables. If the fast path is taken to the end, some of these assignments might not be necessary and can be eliminated using copy propagation. However, these assignments might be necessary if the fast path aborts. This can be addressed by using lazy execution. By delaying these assignments until the points where the fast path aborts, we can safely remove those assignments in the middle of the fast path and improve performance.

5. IMPLEMENTATION

To demonstrate the techniques described in this paper, we have implemented the automatic fast path generation in our ESP [21] compiler. ESP is a concurrent domain-specific language designed to write firmware for programmable devices. It supports the set of language constructs described in Section 2. The ESP language has a number of interesting features. However, those features are orthogonal to the techniques described in this paper.

The ESP compiler uses the process-based approach to generate code. We modified the compiler to add support for fast paths. The symbolic execution to extract fast path is implemented late in the compilation process (right before code generation). Once the fast path is extracted, the traditional optimizations (that were already implemented in the compiler) are applied to it. We also implemented the automata-based approach in the compiler. This currently handles only a subset of ESP programs, namely filters (Section 6.1).

We are still in the process of implementing the preprocessor that translates the fast path specification Section 3 into annotations to the program. Consequently, to perform the experiments described in the paper, we manually added

⁷In these programs, a process is the unit of modularity.

the annotations to the source code to specify the fast path. The annotations added are the same as what the preprocessor would generate. It should be noted that this unimplemented portion has to do with front-end parsing that is fairly straightforward. All other modules that generate and optimize the fast paths have been fully implemented in the compiler. The original ESP compiler has about 7,000 lines of SML code. Our implementation of fast path extraction and optimization required an additional 5,700 lines of code.

6. EVALUATION

In this section, we evaluate the effectiveness of the techniques presented in this paper by applying them to filter programs and to VMMC firmware. In each case, our experiments demonstrate three key points:

1. The programmer effort (annotation complexity) needed to specify the fast paths is small.
2. The automatically extracted fast paths improve the performance of the program significantly.
3. The fast path extraction technique does not increase the size of the executable significantly. Recall that the automata-based approach can lead to an exponential increase in the size of the executable [15].

6.1 Filter Programs

We implemented the four filter programs including 3 used by Probsting *et al.* [29] using our ESP language. These programs perform little actual computation and therefore emphasize the time spent in the process switching code. This makes them ideal to evaluate the effectiveness of automatic fast path construction when compared to fast paths that are manually extracted by the programmer.

A filter program is composed of filters where each filter is a process that reads its inputs from at most one channel and writes its output to at most one channel. A filter program does not allow a filter to wait on more than one channel (i.e. does not support the `alt` statement). The filters are composed linearly such that data flows from the source filter (with no input channel) to the sink filter (no output channel) through a number of intermediate filters. An intermediate filter transforms the data as it flows through it.

Fast Paths. The filter programs involve two stages. The first stage involves performing normal processing on the data passing through it. The second stage involves special case processing when the stream of data ends. In this case, a special value (0) is sent from the source to the sink to signal the end allowing each filter to perform actions that need to be done at the end of the data stream. The first stage is the common path while the second stage executes only once at the end. Therefore, in our experiments, the fast path captures the first stage and leaves the second stage in the slow path.

Annotation Complexity. We wrote the fast path specifications for the three filter programs. The sizes of the four programs (in ESP) are 153, 125, and 190, and 196 lines. The sizes of the fast path specification files are 7, 7, 10, and 10 lines. No scheduling decisions needed to be explicitly specified for these fast paths as the default scheduling policy works well for these programs. Writing the fast path specifications were fairly easy and took 5-10 minutes each. For

these simple programs, the manual fast paths took a little longer—about 30 minutes. However, in one instance (Program 4), it took much longer because our initial attempt had a bug. Recall that the fast path specifications are hints and do not affect the correctness of the program. This is not true when the fast paths are constructed manually.

Experimental Setup. The performance results were gathered on a Linux 2.4 server with a 2.66 GHz Pentium 4 processor and 1 GB memory. We measure the performance by timing 50,000 iterations over 10,000 bytes of data as it flows through the filter program one byte at a time.

We compare four versions of the program. The *ESP* version uses the process-based technique without any fast paths. The *ESP with Manual Fast Paths* version uses the ESP version for the slow path but includes manually optimized code written in C to process the fast path case efficiently. The *ESP with Automatic Fast Paths* version uses the ESP version for the slow path and includes fast paths extracted automatically by the compiler using the techniques presented in this paper. The *Automata-based Compilation* version uses the automata-based approach to compile programs similar to Probsting *et al.* [29].

Performance Results. Table 1 shows that the automatically generated fast path can eliminate most of the performance overhead incurred by the ESP version.⁸ However, there is still some performance difference between automatically and manually generated fast paths. Recall that the filter programs are designed to study the concurrency overheads and therefore exaggerate the difference in performance between the different versions. In addition, the manually optimized code for these small programs is close to the optimal. Finally, our ESP compiler is a research prototype—implementing a number of traditional optimizations will further improve the performance of the automatically extracted fast path.

Our automatic fast path technique outperforms the automata-based code on these programs. This is because the fast paths can be more effectively optimized as it is specialized to handle the common case.

Generated Code Size. Table 1 also shows the sizes of the executable (it shows the number of assembly instructions excluding the initialization sequence). This is different from Probsting *et al.* [29] who use the size of the binary executable file. However, we found that the program initialization code can be significant on these small programs and can distort the results. To validate our results, we compared the binary sizes of the executables and found that our results are similar to their findings.

Table 1 demonstrates that the size of the program does not increase significantly even when automatically extracted fast paths are used. This is because the fast paths extraction technique applies the automata-based approach selectively on certain code paths.

It also shows the automata-based approach can significantly increase the size of the generated code (up to 4 times

⁸The numbers presented here differ from those presented by Probsting *et al.* [29]. The ESP version is significantly slower because the ESP compiler has to handle more general language constructs than just filter programs.

Program	ESP		ESP with Manual Fast Paths		ESP with Automatic Fast Paths		Automata-based Compilation	
	Time	Size	Time	Size	Time	Size	Time	Size
Program 1	36.26	576	2.55	659	2.85	770	3.83	1248
Program 2	85.82	443	1.94	509	2.32	680	3.03	927
Program 3	152.62	611	2.66	709	3.92	795	6.49	3378
Program 4	108.78	963	8.00	1127	15.44	1214	26.06	4956

Program 1 : *ReadFromArray* → *Evener* → *2ByteSwap* → *CRC32* → *WriteToArray*

Program 2 : *ReadFromArray* → *RLE* → *2ByteSwap* → *PES* → *WriteToArray*

Program 3 : *ReadFromArray* → *RLE* → *2ByteSwap* → *PES* → *PES* → *2ByteSwap* → *RLD* → *WriteToArray*

Program 4 : *ReadFromArray* → *CRC32* → *2ByteSwap* → *PES* → *PES* → *2ByteSwap* → *RLD* → *WriteToArray*

Table 1: Filter Programs. *Time* refers to execution time in seconds. *Size* shows the size of the executable in terms of the number of lines of assembly instructions. The filter sequence for the three programs is shown below the table.

in these programs). An earlier study [15] found that the automata-based approach can result in code that is 2 to 3 orders of magnitude larger than that produced by the process-based approach.

6.2 VMMC Firmware

In this section, we demonstrate the effectiveness of compiler generated fast paths on VMMC firmware that runs on a network card. The Virtual Memory-Mapped Communication (VMMC) architecture [14] delivers high performance on gigabit networks by using sophisticated network cards. It allows data to be directly sent to and from the application memory (thereby avoiding memory copies) without involving the operating system (thereby avoiding system call overhead). The operating system is usually involved only during connection setup and disconnect. The VMMC implementation [14] currently uses the Myrinet [7] network interface cards.

The VMMC firmware was implemented first using event-driven state machines in C, and then reimplemented using ESP. The C version of the VMMC implementation includes about 15,600 lines of C code; around 1,100 of these were used to implement the fast path. The ESP version of the code has about 500 lines of ESP code together with around 3,000 lines of C code. The VMMC firmware is reasonably complex and allows us to evaluate our technique in a more realistic scenario.

Fast Paths. We added three fast paths in the ESP code: two in the data send operation for normal size (≥ 64 bytes) and small size (< 64 bytes) respectively, and one in the data receive operation. The fast paths we selected are similar to the manual fast paths implemented in corresponding C versions. However, one of the manual fast paths in the C implementation includes an optimization that our compiler cannot automatically perform. This optimization accounts for the difference in the performance between the two hand-optimized C versions in the Latency microbenchmark for packets larger than 64 bytes.

The one optimization not implemented in the automatically extracted fast path requires a nonfunctional transformation. This optimization in the C fast path involves using a different algorithm to perform “data pipelining”⁹ on the

⁹ The “data pipelining” technique decreases the latency for the first page (4 Kbytes) of data and works as follows: When the VMMC program is sending out messages, it need to

fast path. This algorithm is difficult to express in modular slow path code. Consequently, a simpler algorithm is employed in the slow path code. Since this fast path is not functionally equivalent to the original code, the techniques described in the paper cannot be used for it. We plan to explore this in the future.

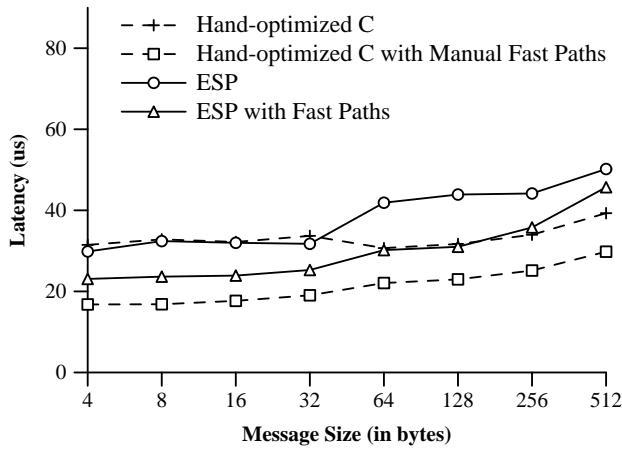
Annotation Complexity. The specification for three fast paths in our VMMC program has 20, 14, and 18 lines respectively. They were easy to write and required 10-20 minutes each. In contrast, the 1,100 lines of fast path code in the C implementation that were manually implemented took several months of writing, optimizing, and debugging. The automatic fast path extraction technique described in this paper reduced the programmer effort by orders of magnitude when compared with the manual approach.

Experimental Setup. All experiments measurements use a pair of PC. Each PC has a 300 MHz Pentium processor, 128 MB memory and a Myrinet network interface card. The nodes are directly connected to each other using a Myrinet cable. The PCs run Windows NT 4.0.

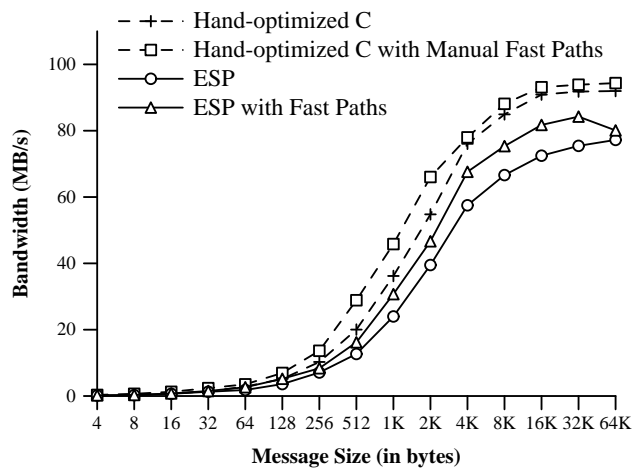
The VMMC firmware runs on the Myrinet network cards which have a programmable 33-MHz LANai4.1 processor, 1 Mbyte SRAM memory and three DMA engines to transfer data— one to transfer data to and from the host memory; one to send data out onto the network; one to receive data from the network. Myrinet is a packet-switched gigabit network. The Myrinet network card is connected to the network through two unidirectional links of 160 Mbytes/s peak bandwidth each. The actual node-to-network bandwidth is usually constrained by the PCI bus (133 Mbytes/s) on which the network card sits.

We compare the performance of four versions of the VMMC firmware. The *ESP* version implements the firmware in ESP. The *ESP with fast paths* version includes optimized fast paths generated by the compiler. The *hand-optimized C* version is an optimized firmware implementation in C. This version uses event-driven state machines as the con-

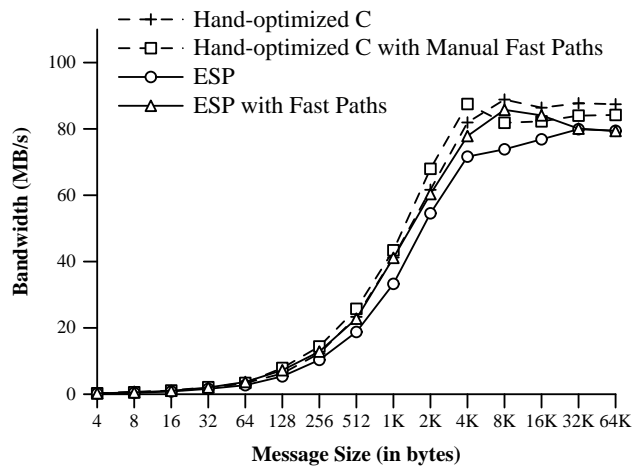
first transmit the data from user memory to the card via DMA before it can send the data on the network. In the C fast path, as soon as a certain amount of new data arrives, the program begins the transmission on the network. Then it keeps polling between the DMA engine and the network transmit engine to transmit the remaining pieces as soon as they arrive.



(a) Latency Microbenchmark



(b) One Way Bandwidth Microbenchmark



(c) Bidirectional Bandwidth Microbenchmark

Figure 7: Microbenchmarks' Performance. The graphs have discontinuities at the 32/64 byte boundary as well as at 4/8Kbyte boundary. The former is because small messages (≤ 32 bytes) are handled differently. The latter is because the page size is 4Kbytes. Requests that span multiple pages are broken down into multiple transfers.

currency primitive and is structurally similar to the ESP implementation [21]. The *hand-optimized C with manual fast paths* version includes fast paths described above that were manually extracted and highly optimized.

The automata-based compiler that we implemented is not general enough to handle the VMMC firmware. In Section 6.1, we demonstrated that the code size increase even for small filter programs can be substantial. Due to the limited memory available on the network card, we believe that the code generated by the automata-based approach is likely to be too big to run on the network card. Even though the network card has 1MB of memory, most of it is reserved for the data buffers—only a small portion is reserved for the instructions.

Three microbenchmarks are used to compare the performance of the four versions of the VMMC firmware. Each microbenchmark is run on a pair of machines that communicate with each other using VMMC. The *Latency* microbenchmark measures the latency of sending a message of a particular size between two machines. This is measured using a simple pingpong program that sends a message back and forth between the two machines. The *Bandwidth* microbenchmark measures the bandwidth that can be achieved between two machines when sending messages of a particular size. This is measured by using a program on one machine to continuously send messages of that size to a program on the second machine that is repeatedly receiving the messages. The *Bidirectional Bandwidth* microbenchmark measures the total bandwidth between two machines when both machines are sending messages of a particular size simultaneously.

Performance Results. Figure 7 presents the microbenchmark performance. In each case, the x-axis shows the message size.

In the Latency benchmark, the latency for 4 byte to 32 byte messages dropped from around $31 \mu s$ to about $24 \mu s$, which is a 22% improvement in performance. As noted earlier, some of the latency difference between the ESP version and the C version with fast paths for larger packet size (512 bytes) is due to the “data pipelining” that is currently not implemented in the ESP version.

In the Bandwidth benchmark, adding the fast path also helped in closing the performance gap between ESP version and C versions. For example, for 64 byte messages, the bandwidth increased from 1.8 MB/s to 2.6 MB/s, which is about 40% improvement. For 512 byte messages, the bandwidth increased from 12.6 MB/s to 16.2 MB/s, which is an improvement of about 28%. For 2 Kbyte messages, the bandwidth increased from 39.5 MB/s to 46.7 MB/s, which is about 18% better.

Generated Code Size. The size of the executable generated for the ESP version is significantly smaller than the C version even after fast paths are added. The C version with manual fast paths has 40,732 lines of assembly instructions in the executable. The ESP version without fast path has 12,935 lines of assembly instructions while the version with fast path has 22,480 assembly instructions in the executable.

7. RELATED WORK

Fast Paths. A number of research projects [30, 26, 23] investigate the use of fast paths in sequential programs.

The Synthetix project [30] manually generated fast paths in the HP-UX operating system. Since manual program specialization is time consuming and error-prone, they have developed a toolkit and methodology that allows the programmer to systematically specialize system software [25].

The Scout operating system [26] makes path an explicit abstraction mechanism to improve resource allocation and scheduling decisions. It uses compiler optimizations like outlining, cloning, and path inlining to improve the performance of the fast paths [27].

Formal methods can be used to build optimized fast paths [23] in the Ensemble network architecture [18]. A protocol stack consists of a sequence of protocol layers. The NuPRL [13] system was used to semi-automatically extract a fast path from the protocol stack.

Automata-Based Approach to Compiling Concurrent Programs. A number of concurrent languages have compilers that compile a concurrent program to run efficiently on a single processor.

Esterel [6] is a synchronous language designed to model the control of concurrent systems. Earlier Esterel compilers [6, 12] used the automata-based approach to generate code. More recently, gate-based compilers [5] have been implemented. They avoid the code blowup using the automata-based compiler but incur a significant runtime overhead. Process-based compilers [15] have also been implemented for Esterel. However, they can handle only a subset of valid Esterel programs—those in which a valid schedule for the concurrent Esterel program can be determined statically.

Edwards *et al.* [15] evaluates the tradeoff of using each of the three approaches—automata-based approach, gate-based approach, and process-based approach—for compiling Esterel programs. As expected, the automata-based compiler [6] generates the fastest code but the size of the executable can be 2–3 orders of magnitude larger than the other approaches. The process-based approach generates code that is only twice as slow as the automata-based approach but yields the smallest executables.

Squeak [10] uses the automata-based approach to generate sequential code. It considers all possible interleavings of the concurrent program. At each stage, one of the unblocked processes is executed for one step. A random number generator is used to select a process when multiple processes are ready for execution.¹⁰ Filter Fusion [29] uses the automata-based approach to fuse filters (Section 6). A sequential program is obtained by successively fusing pairs of adjacent filters into a single filter using a technique similar to that used in Esterel compilers [6]. The StreamIt compiler [17] also uses fusion based techniques to compile concurrent programs.

Other. There is a large body of work that focuses on re-ordering basic blocks of instructions since the seminal work in the area by Fisher [16]. The idea is to create large su-

perblocks of instructions using a sequence of basic blocks that are usually executed consecutively. The superblock can then be more effectively optimized. In the case when a superblock is exited prematurely, some extra work might have to be done to patch up the program state to correctly handle it. In this respect, it is similar to fast path generation where the fast paths can abort prematurely.

Ammons *et al.* [1] show that data-flow analysis can be made more effective by applying them to the commonly used paths in isolation. They identify and duplicate the commonly used paths and show that constant propagation works better on these paths because they do not have to deal with the infrequently used paths. They focus on acyclic paths in sequential programs. In contrast, the work described in this paper handles paths that can include cycles and span multiple processes in a concurrent program.

Path Expressions. Path expressions [9] were originally proposed to specify synchronization between processes. The path expressions specified a set of legal ordering of accesses to a shared resource. If a request to access the resource does not conform to the order specified by the path expression, the requesting process would block until the concurrent program reached a state that allowed the blocked process to continue.

Path expressions have been extended and widely used to describe data and control flow of a program. For instance, Generalized Path Expressions [8] have been used to debug sequential programs. They specify valid execution paths in the program in terms of program events and variables. Bugs in the program can be identified by using the path expression to query the execution trace of the program.

In this paper, we have extended path expressions to specify fast paths through a concurrent program.

8. CONCLUSIONS

This paper presents a technique that automatically extracts fast paths from concurrent programs. The compiler uses fast paths specification provided by the programmer to extract fast path. It uses symbolic execution to isolate and aggressively optimize the fast path code while using less aggressive techniques for the rest of the program. Our experiments demonstrate that our approach improves performance significantly without blowing up the size of the generated code.

The use of fast paths in concurrent programs allows us to narrow down the performance gap between the process-based and automata-based compilations without suffering from the exponential code size increase that can result from using automata-based approach. Automatic fast path extraction avoids the errors introduced during manual optimizations and simplifies the process of fast path construction. Therefore we believe that this technique would prove very useful in writing high performance concurrent programs for embedded devices.

Acknowledgments

This project is supported in part by DOE grant DE-FC02-01ER25456, by NSF grant EIA-0101247 and ANI-9906704, and by a grant from the Intel Research Council.

¹⁰In contrast, Esterel programs are deterministic—all possible schedules yield the same result. Therefore, it does not require a random selection at each stage.

9. REFERENCES

- [1] G. Ammons and J. Larus. Improving Data-flow Analysis with Path Profiles. In *Programming Languages Design and Implementation*, 1998.
- [2] G. R. Andrews. *Concurrent Programming*. Benjamin/Cummings Publishing Company, 1991.
- [3] T. Ball and J. Larus. Efficient Path Profiling. In *IEEE Micro*, 1996.
- [4] A. Basu, T. von Eicken, and G. Morrisett. Promela++: A Language for Correct and Efficient Protocol Construction. In *Infocom*, 1998.
- [5] G. Berry. *The Constructive Semantics of Pure Esterel*. Draft 3, 1999.
- [6] G. Berry and G. Gonthier. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2), 1992.
- [7] N. J. Boden, D. Cohen, R. E. Felderman, A. E. Kulawik, C. L. Seitz, J. N. Seizovic, and W.-K. Su. Myrinet: A Gigabit-per-Second Local Area Network. *IEEE Micro*, 15(1):29–36, 1995.
- [8] B. Bruegge and P. Hibbard. Generalized Path Expressions: A High-Level Debugging Mechanism. *Journal of Systems and Software*, 3:265–276, 1983.
- [9] R. H. Campbell and A. N. Habermann. The specification of process synchronization by path expressions. *Lecture Notes in Computer Science*, 16:89–102, 1974.
- [10] L. Cardelli and R. Pike. Squeak: A Language for Communicating with Mice. *Computer Graphics*, 19(3):199–204, July 1985.
- [11] C. Castelluccia, W. Dabbous, and S. O'Malley. Generating Efficient Protocol Code from an Abstract Specification. In *SIGCOMM*, 1996.
- [12] M. Chiodo, P. Guisto, A. Jurecska, L. Lavagno, H. Hsieh, K. Suzuki, A. L. Sangiovanni-Vincentelli, and E. Sentovich. Synthesis of Software Programs for Embedded Control Applications. In *Design Automation Conference*, 1995.
- [13] R. L. Constable, S. F. Allen, H. Bromley, W. Cleaveland, J. Cremer, R. Harper, D. J. Howe, T. Knoblock, N. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Development System*. Prentice-Hall, 1986.
- [14] C. Dubnicki, A. Bilas, Y. Chen, S. Damianakis, and K. Li. VMMC-2: Efficient Support for Reliable, Connection-Oriented Communication. In *Hot Interconnects*, 1997.
- [15] S. A. Edwards. Compiling Esterel into sequential code. In *Design Automation Conference*, 2000.
- [16] J. A. Fisher. Trace Scheduling: A Technique for Global Microcode Compaction. *IEEE Transactions on Computers*, C-30:478–490, 1981.
- [17] M. I. Gordon, W. Thies, M. Karczmarek, J. Lin, A. S. Meli, A. A. Lamb, C. Leger, J. Wong, H. Hoffmann, D. Maze, and S. Amarasinghe. A Stream Compiler for Communication-Exposed Architectures. In *Architectural Support for Programming Languages and Operating Systems*, 2002.
- [18] M. Hayden. The Ensemble System. Technical Report TR98-1662, Computer Science Department, Cornell University, 1998.
- [19] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, Aug. 1978.
- [20] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable Stream Processors. *IEEE Computer*, August:54–62, 2003.
- [21] S. Kumar, Y. Mandelbaum, X. Yu, and K. Li. ESP: A Language for Programmable Devices. In *Programming Languages Design and Implementation*, 2001.
- [22] J. Larus. Whole Program Paths. In *Programming Languages Design and Implementation*, 1999.
- [23] X. Liu, C. Kreitz, R. van Renesse, J. Hickey, M. Hayden, K. Birman, and R. Constable. Building Reliable, High-Performance Communication Systems from Components. In *Symposium on Operating Systems Principles*, 1999.
- [24] C. D. Marlin. Coroutines – A Programming Methodology, a Language Design and an Implementation. *Lecture Notes in Computer Science*, 95, 1980.
- [25] D. McNamee, J. Walpole, C. Pu, C. Cowan, C. Krasic, A. Goel, P. Wangle, C. Consel, G. Muller, and R. Marlet. Specialization Tools and Techniques for Systematic Optimization of System Software. *Transactions on Computer Systems*, 19(2):217–251, 2001.
- [26] D. Mosberger and L. L. Peterson. Making Paths Explicit in the Scout Operating System. In *Operating Systems Design and Implementation*, 1996.
- [27] D. Mosberger, L. L. Peterson, P. G. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. In *SIGCOMM*, 1996.
- [28] R. Pike. The implementation of newsqueak. *Software, Practice and Experience*, 20(7):649–660, 1990.
- [29] T. A. Proebsting and S. A. Watterson. Filter Fusion. In *Principles of Programming Languages*, 1996.
- [30] C. Pu, T. Autrey, A. Black, C. Consel, C. Cowan, J. Inouye, L. Kethana, J. Walpole, and K. Zhang. Optimistic Incremental Specialization: Streamlining a Commercial Operating System. In *Symposium on Operating Systems Principles*, 1995.
- [31] M. Rajagopalan, S. K. Debray, M. A. Hiltunen, and R. D. Schlichting. Profile-Directed Optimization of Event Based Programs. In *Programming Languages Design and Implementation*, 2002.
- [32] R. von Behren, J. Condit, and E. Brewer. Why Events are a Bad Idea (for high-concurrency servers). In *Hot Topics in Operating Systems*, 2003.