UNIVERSITY OF CALIFORNIA, SAN DIEGO

ASIC life extension through hardware patch interfaces

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Computer Science

by

Vladyslav Sergeevich Bryksin

Committee in charge:

Professor Steven Swanson, Chair
Professor Michael B. Taylor
Professor Dean M. Tullsen

2009

The Thesis of Vladyslav Sergeevich Bryksin is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

Chair

University of California, San Diego

2009

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

First, I'd like to thank my parents for trying their best to make a decent individual out of an airhead that I was. I thank my advisors, Steven Swanson and Michael Taylor for providing me with advice, guidance and inspiration. Also, I thank Dean Tullsen for contribution of his time to serve on my thesis committee. This work would not be possible without the contributions of fellow labmates: I thank Nathan Goulding for his infinite kindness and help, Ganesh Venkatesh for his help in development of the patching algorithm, Jack Sampson, who has been essential in long discussions about design issues and impromptu opera performances, and Jose Lugo-Martinez for his supplies of Puerto Rican rum. Finally, I would like to thank my girlfriend, Zhenya, and all of my friends for their support and existence.

ABSTRACT OF THE THESIS

ASIC life extension through hardware patch interfaces

by

Vladyslav Sergeevich Bryksin
Master of Science in Computer Science

University of California San Diego, 2009

Professor Steven Swanson, Chair

Specialized processor designs and ASICs offer lower power consumption and greater efficiency compared to general purpose processors. However, the drawback of specialized hardware designs is the reduction in the generality of workloads that they are able to handle.

An important characteristic of specialized hardware designs is the inability to manage changes in the underlying applications. This thesis describes and analyzes the concept of ASIC patching in the Arsenal design: a mechanism to mitigate the effects of software evolution for the ASIC that preserves the benefits given by the specialized hardware. The code changes between versions can be handled by augmenting ASIC with configurable hardware gadgets and mechanisms to transfer control flow from the ASIC to a specialized patch processor that executes the code fragments that differ between versions. Thus, the lifetime of the ASIC is extended by abstraction of code changes from the original application into patches, and execution of these patches through one of the proposed patching mechanisms.

The results show that majority of changes in the application test set are amenable to patching, and that hardware patching mechanism proposed in this thesis is a viable approach that can handle a wide range of changes in the underlying application code with reasonable performance overhead.

1 Introduction

1.1 Building patchable specialized processors

Specialized processor designs and ASICs offer lower power consumption and greater efficiency compared to general purpose processors. However, the drawback of specialized hardware designs is the reduction in the generality of workloads that they are able to handle.

Arsenal design aims to achieve performance gains and lower power consumption over a general purpose processor by incorporating 10s to 1000s of specialized processing elements (SPEs) into one system. These SPEs represent a variety of hardware designs, from general purpose processors to specialized processors and ASICs that handle a particular functionality (i.e. DSP algorithms, encryption algorithms, graphics accelerators). The architecture of the Arsenal system is described in Section 2.

This thesis describes and analyzes the concept of SPE patching in the Arsenal design: a mechanism to mitigate the effects of software changes for the SPEs that preserves the benefits given by the specialized hardware. Thus, the patching process abstracts the changes in the application into a patch that describes how a particular change is to be handled by the Arsenal runtime; the runtime, in turn, determines the appropriate method for execution and configures the SPE with this patch.

An important aspect of specialized hardware designs is the inability to manage changes due to the software evolution in the underlying applications. Thus, faced with a new version of an application that is targeted by a given SPE, the existing approach in the Arsenal toolchain is to designate this SPE as unsuitable

for this version, and execute it on the general purpose processor. While this is a viable technique that does not forfeit generality, the benefits of power savings and potential speedup from executing on a specialized design are reduced. Moreover, the analysis of changes in the software, described in Sections 1.2 and 5.1, shows that there exists an ample set of changes between versions that does not require a fall back on the execution on the general purpose processor. These changes can be handled by augmenting SPEs with configurable hardware gadgets and mechanisms to transfer control flow from SPEs only to execute the parts that are different between versions.

The focus of this thesis is not a recovery from design bugs, but adapting the SPEs to be resilient to changes in the future versions of the underlying applications. While the generality of the types of changes that a patch process can handle is an important characteristic, there is a trade-off between generality and the performance, power and area aspects of SPEs. On one side of the extreme, the incorporation into each SPE a way of handling any type of change in place implies an incorporation of a general purpose processor core into each SPE, which eliminates the power and area savings of the Arsenal design. On the other side, a software fall back mechanism on the general purpose processor means that a single insignificant change can render an SPE unsuitable for a current version of the application. Thus, one of the goals of the patching process is to find a proper balance between the performance and power and area for the patching mechanisms. Also, another goal is to establish a balance between sufficient coverage of types of changes for a given patch mechanism and the overhead that this coverage introduces for each mechanism. For a given lifetime of a Arsenal system, the SPE algorithms might change, and the right balance of these requirements would exhibit the graceful degradation of performance of the system, where the initial changes can be handled with insignificant overhead, while cumulative subsequent changes to the application over time require fall back on a less efficient, but more general patching mechanisms.

This thesis explores the classification of common changes between software versions, design and examination of patching methodologies in the Arsenal

toolchain, and compares the resulting patchable Arsenal system with the baseline system.

1.2 Identifying changes in software between versions

In order to be able to generate SPEs that are able to handle application changes and display graceful degradation in performance in face of these modifications, the common types of software changes need to be identified. While it is impossible to foresee the changes in the applications, an analysis of a set of applications in their respective domains gives an outlook on the types of changes that are common and provides an understanding on the patterns of software evolution. The chosen set of applications reflects the scope that is applicable to the purpose of SPEs: DSP applications, encoding and decoding, encryption, and other applications that are amenable to hardware instantiations and spend major fraction of their time executing inside loop bodies. These applications are well known, have stable release cycle and include Lame MP3 Encoder [7], an MPEG Audio Layer III encoder project established in 1998, libjpeg [8], a JPEG encoder/decoder, and bzip2 [9], a data compression/decompression project that was first released in 1996.

To quantify the amount of changes in software, several metrics important to the scope of this thesis were used. In the context of the SPEs, the instruction level analysis of differences allows to identify the corresponding hardware basic blocks that change accordingly. The changes were broken down into several categories: data flow, control flow, interface and language changes. Each of these categories was further refined into smaller categories to target specific constructs, and Section 5.1 describes the results of this study. Based on these results, we identified and developed patching methodologies that are described in Section 4.

2 Background

The Arsenal architecture aims to address the issues of power and architectural scalability due to threshold voltage scaling limitations. While a transition to multi-core designs is able to mitigate these issues, the problem of utilization of the whole chip at full frequency will re-emerge in a few generations of process scaling. The approach that allows to overcome these issues while gaining in performance is to vary the parts of the die that are active during runtime by combining an array of massively heterogeneous processors into one system. The diversification of the specialization of the processors allows to target only specific applications and use a fraction of a chip at once. The Arsenal system is comprised of processors that range from general purpose processors to SPEs that target specific applications and software constructs. This specialization allows to improve power efficiency and performance while providing a way to execute general purpose applications.

The Arsenal processors are organized into clusters that contain a variable number of SPEs of mixed sizes and specialization. Figure 2.1 shows an Arsenal processor consisting of twelve SPE clusters connected to four banks of L2 cache via a grid-based interconnect.

While the sizes of the SPEs in the cluster depend on the architectural properties of the SPE and the available area budget in the cluster, the grouping of the SPEs is determined by the common functionality. Since SPEs that have related functionality, or can be combined to perform some functionality are likely to be used together, the co-location of them in the same cluster has a benefit of sharing the same L1 cache. The inter-cluster grid-based interconnect interfaces the clusters or complexes with L2 caches and general purpose processors, which is similar to the designs described in the tiled processors such as RAW [12] and

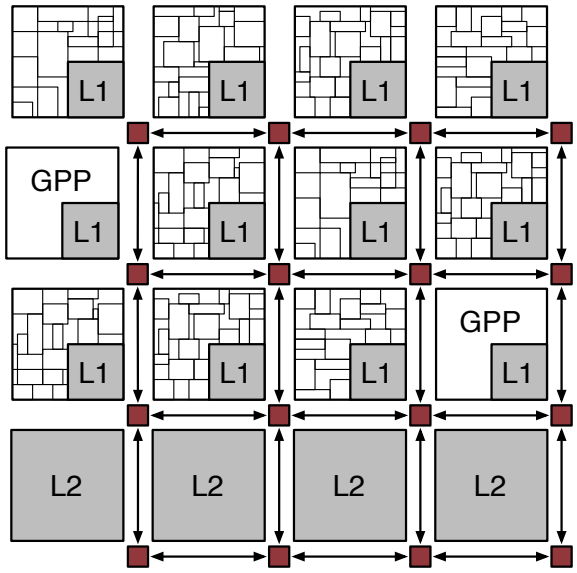


Figure 2.1: An Arsenal processor.

WaveScalar [13]. The intra-cluster interconnect connects individual SPEs to the inter-cluster grid with the constraint that only one SPE within a cluster can be active at a time. This constraint is only valid for intra-cluster scheduling, while SPEs from different clusters can execute at the same time.

The memory hierarchy contains L1 caches that are allocated within each cluster, L2 caches allocated per a set of clusters, main memory, and optional L0 caches that are local to each SPE. A MESI-based coherency protocol ensures coherency within all L1 caches, L2 and the main memory, and a release consistency model is used for the communication between processors through shared memory.

A per application analysis identifies the mappings of the regions of code to the available SPEs in the system using several metrics that include the profile of the workload, the performance data for the execution on suitable SPEs, communication between the regions of code and the power constraints on the workload. The runtime of the Arsenal processor maps the code to be executed to a set of the available SPEs and schedules them dynamically by evaluating the contention for a given SPE, its locality, and the runtime behavior of the application. Thus, migration of the application is present both statically, by identifying the sequence of the suitable SPEs, and dynamically, when the runtime status of the system

determines the set of possible SPE choices.

SPE generation in the Arsenal toolchain is a current focus of our group, where some SPEs such as general purpose processors and established designs can be synthesized from existing solutions, while special purpose processors are synthesized automatically from high level languages such as C or C++. This automatic synthesis approach is based on transformation of the applications into program dependence graph form [14] and merging isomorphic components of the graph. Thus, the common workloads are analyzed and semantically similar parts are merged to produce more generic SPEs given the area and power limitations, while reducing the overall amount of hardware for the Arsenal processor.

3 Related work

The problem of hardware patching is explored in several works, however, the emphasis of these papers are patching hardware bugs that are not caught during testing and validation. The authors of [2, 3, 4] propose an inclusion of programmable hardware in the chip that is able to handle exceptional conditions by tapping onto key logic signals and comparing these signals to known errata signatures. Upon encountering a defect, an appropriate action is performed that can range from flushing a pipeline to dynamic instruction stream patching, replay after checkpoint or flush, or invocation of the programmable error handler. These papers address the avoidance and recovery of certain conditions that arise due to the hardware bugs, and do not address the extension of the functionality and evolution aspects of hardware.

Ginseng, a dynamic software update implementation for C language [6], shows the software approach of dealing with applications evolution and patches. The authors show that, in principle, any program can be patched while running without the need to stop and restart this application. Ginseng compiler compares the current version with new releases and generates patches that can be uploaded during runtime. While this approach does not require any hardware modifications and therefore is not applicable to hardware patching, it shows that dynamic software patching process is practical and guarantees type safety and data coherence.

The authors of RAMA [5] demonstrate a configurable datapath for Systems on Chip (SoC), where reconfigurable busses between DSP blocks on a MIMD DSP array architectures are introduced. Thus, the granularity of configuration are DSP blocks that are dynamically combined in a datapath to implement a variety of DSP applications, and the configuration is performed on the inter-block level. While the

problem of inter-block configuration is focused on in the Arsenal architecture, this thesis explores the problem of intra-block level configuration where the granularity of data flow changes can be on the level of basic blocks of the SPEs.

VEAL, the Virtualized Execution Accelerator for Loops [10], proposes a generalized loop accelerator whose main goals are a cost and area effective generalization of loop constructs with an abstracted interface that allows to dynamically map loop bodies to an available accelerator. The tradeoffs between area and performance described in this paper are representative of the balance that this thesis aims to achieve: sufficient coverage of common code patterns given limited area without generalization to a general purpose processor.

4 Description of hardware patching methodologies

Patching methodologies depend on the amount and impact of changes in the underlying software that can be accommodated by the hardware. An orthogonal issue to the ability to handle various changes is the tradeoff between performance, area and power devoted to each mechanism. While delegating patching to the general purpose processor allows a general way to handle all types of changes since arbitrary code can be executed, other patching methods offer less general but more lightweight route to manage application changes.

Without the patching mechanism, if there is a change in the application for a given SPE, this SPE is considered to be unusable for a given software version, and the handling of this SPE functionality is delegated to the general purpose processor. This is due to the fact that SPEs do not support the transfer of control from an arbitrary basic block to another processor and the transfer back to the SPE. Consequently, the transfer of the SPE state information from arbitrary point in the execution and transfer of the new state of this SPE back from the processor to another arbitrary point is not possible without additional modifications. Therefore, a software fall back mechanism is a default way to address changes between application versions in the unpatched version of the Arsenal toolchain.

To enable patching in the Arsenal system, three methodologies are proposed that target different types and magnitudes of changes between application versions: SPE gadgets, Specialized Patch Processor and software fallback.

The first method to handle application changes is to add gadgets to the SPEs to make them more resilient to changes. An SPE gadget is a modification to

modules in the basic blocks that allows a configurable behavior at runtime without the overhead of interrupting an SPE to execute code on another processor.

When faced with changes in the application that cannot be handled by gadgets, an exceptional condition is set on the execution path. With the appropriate changes to the control and data paths of the SPE that are described below, the data that contains the state of the SPE can be transferred between the SPE and a processor. Thus, the SPE can stop execution upon encountering a block of code that it is unable to execute on existing hardware and transfer its state to a general purpose processor. This general purpose processor, in turn, executes the code that is not present in the default application version of the SPE, modifies state of the SPE if it is required, and transfers this state back to the SPE so that it can resume execution. This software fall back mechanism is also feasible in the case of large changes to the application, where the execution of a patched version on the SPE with the overhead of transfers of control between SPE and a general purpose processor exceeds the time to execute the application on the general purpose processor.

While the described software fall back on a general purpose processor design is able to handle arbitrarily changes in the applications, the drawbacks of this baseline mechanism are increased power usage, load increase on a general purpose processor, and potential increase of execution time. A proposed solution is to delegate the patch handling from a general purpose processor to a Specialized Patch Processor (SPP). An SPP is an area optimized general purpose processor that is allocated for a small set of the SPEs within a cluster whose main purpose is the execution of patches.

4.1 Gadgets

While some changes to the applications might trigger an exceptional condition due to inability to execute code for some version of the application and require transfer of control from an SPE, there exists a large subset of changes that can be handled by augmenting the SPE with gadgets. For instance, if the offset of a field

in the structure has changed, a gadget that replaces constant offsets in the SPE with configurable registers to hold the offset of the field allows to set a correct offset for each version of the application. Thus, the execution of the SPE that is augmented with gadgets does not incur performance overhead of the context switch on an exceptional condition if this condition can be handled by a gadget. The servicing of this condition by a gadget introduces minimal or none overhead of additional cycles, since gadgets are modifications of existing modules and are a part of the same basic block as the original module. Thus, even more complicated gadgets that take several cycles to execute introduce an overhead that is less than a cost of an interrupt and exception handling on another processor. Nevertheless, gadgets on the critical path of the execution might increase cycle time.

However, the usefulness of gadgets is limited by several factors. First, addition of each gadget results in the increase of the SPE area since each original module is replaced with the module that is augmented with the gadget for all basic blocks. Thus, an important metric for gadgets is the percentage of each type of changes in the corpora of profiled applications that each gadget can manage over the area increase due to the addition of this gadget. Second, the amount of changes that can be handled by gadgets is restricted. Given limited area available for the SPE, it is not feasible to include complex gadgets that can handle arbitrary code execution. Third, configuration of gadgets introduces a performance overhead during the initial configuration of the SPE. For each gadget, certain number of bits needs to be passed to the SPE to set the appropriate configuration for a given application version. This process is described in Section 4.5. Even though the initialization of gadgets incurs an overhead of shifting in the configuration values for each gadget, this initialization is only done when an SPE is configured for a certain version. All subsequent calls to the SPE do not incur this overhead. Moreover, SPE initialization also includes the control path scanchain initialization which is done in parallel with the gadget scanchain.

Given the benefits of gadgets, their range of employment is limited to small changes in the application between versions. Even though this range can be extended to handle a wider scope of changes, the limited area budget would make

this approach impractical.

The gadgets described below are connected to the gadget scanchain and are initialized during the initial SPE configuration described in Section 4.5. During the C to Verilog compilation, the modules that can be converted to gadgets are identified and replaced. Given the results of software differences study described in Section 5.1, we analyzed the categories that contained the most changes to identify the ways to make SPEs more resilient to software evolution. While the gadgets proposed in this thesis target the categories with the most changes, we anticipate a greater variety of gadgets that target more refined classes. An important aspect of gadget inclusion is the impact on the area and the performance of the SPE that is shown in Section 5.2. Thus, a viable direction of gadget design is in-depth instead of in breadth, that is, to focus more on the efficiency, configurability and reusability of gadgets instead of creating a wide array of gadgets that can handle limited functionality.

4.1.1 Configurable offsets for structures

One of the frequent changes in the set of profiled applications is the layout change in the data structures. These changes include addition or removal of data structure elements, reorganization and change of levels of indirection in the data structure. For instance, Table 4.1 shows code changes for structure *Foo*, where variable x is removed, v is added, and y and z are switched.

Function *decode_mcu* in libjpeg shows an example of change of levels of indirection between versions 5 and 6:

- s += state.cur.last_dc_val[ci];

+ s += state.last_dc_val[ci];

All offsets for arrays and other data structures in the SPE are converted into configurable registers during the C to Verilog compilation, and are currently set to be 32 bit. Figure 4.1 shows a block diagram of a structure access with baseline and gadgetized modules.

While not a design feature of this gadget, configurable offsets allow to reference any location on the local stack by manipulating the offset.

Table 4.1: A sample structure layout that changes between versions

Original code	New code
struct Foo{	*struct Foo{*
int x;	*int new;*
int y;	*int z;*
int z;	*int y;*
*Foo *next;*	*char *v;*
};	*Foo next;*
	};

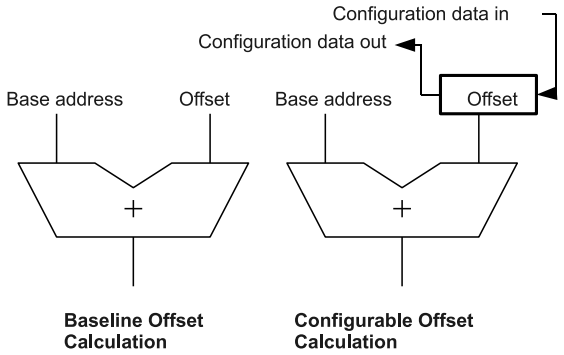


Figure 4.1: Data structure calculation with configurable offset register.

4.1.2 Simple ALU for each operation

Another frequently observed type of changes between software versions is the change of operators. The most common occurrence of operator changes is in the loop construct operators, conditional statements, and less frequent in the data flow operator changes. For instance, loop array bounds and operators are changed in function *qdescale_zig* between libjpeg versions 2 and 3:

- for (i = 0; i < DCTSIZE2; i++) {
+ for (i = DCTSIZE2 - 1; i >= 0; i–) {

Conditional statement operators change in function *HuffmanCode* between versions 3.92 and 3.93 of Lame:

- if (x1 < 0) {
+ if (x1 != 0) {

And the data flow operators change in function get_bits between versions 1 and 2 of libjpeg:

- get_buffer = (get_buffer << 8) + c;
+ get_buffer <<= 8;
+ get_buffer |= c;

The proposed gadget is a replacement of all single operator modules with a simple ALU that does not include multiply and divide operations.

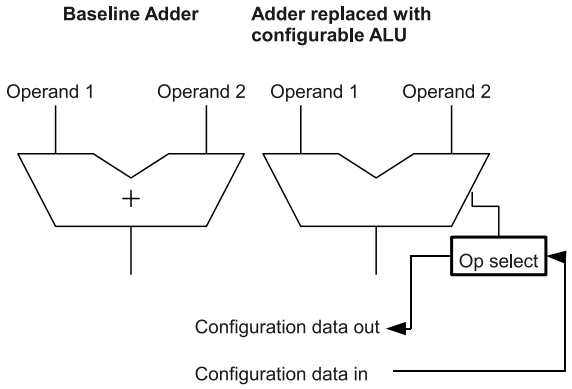


Figure 4.2: An adder and a configurable ALU.

In the block diagram in Figure 4.2, the adder module is replaced with the ALU gadget, and the ALU operation select is a 4-bit register that is connected the gadget scanchain.

4.1.3 Configurable constants

In addition to changing the operators for conditional statements and loops, the replacement of all constant values with configurable registers allows fully configurable loop and conditional statements constructs for the cases where constants are used as loop bounds and compare targets. For instance, a simple loop construct

for (i = 0; i < MAX_N; i++)

will have an ability to set constants 0, MAX_N and operations "<" and "++" to arbitrary values.

4.2 Datapath register placement

Before the description of marshalling gadget, the following sections describe the placement of the registers to store state of the SPE's datapath, and the implementation of scanchains.

The SPE is divided into control path, which represents the state machine of the SPE, and datapath, which operates on the data. The basic blocks in the datapath correspond to the states in the control path, where each basic block takes one to several cycles to execute. The inputs and outputs of the basic block are the sets of variables that are live accross this basic block, and the values of these variables are preserved in registers on the basic block boundaries.

The initial design of SPEs in the Arsenal toolchain used register per live variable allocation scheme. Thus, for each live variable in each basic block of the datapath, if a variable was live in the end of this block, its value was latched to a live-out register corresponding to that block. This allocation scheme had a property that at any point of execution, all live variables state was available in registers of a current basic block. However, this lead to a large number or registers that were never accessed. Figure 4.3 shows a diagram of three register placement designs, where variable x is defined in basic blocks 2 and 3, and used in block 4, variable y is defined in block 1 and used in blocks 2 and 3, and variable z is defined in block 4.

An important optimization for the state manipulation was the reduction

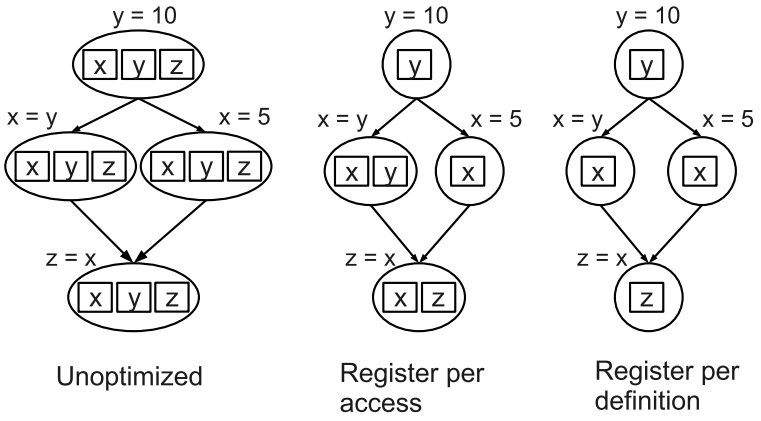


Figure 4.3: A block diagram of 3 register placement designs: unoptimized, register per access and register per definition.

of the state via elimination of redundant registers. For instance, if variable x is defined in basic block 1, and written in basic block 10, the registers that latch the value of x between these blocks are never accessed and can be replaced with wires.

There are two issues related to register placement in the toolchain. First, whether the registers are considered live-in or live-out for a given basic block, i.e., whether the live variables are latched into registers before or after the execution in the current block. Second, whether a register per definition or register per access allocation scheme is used, where a register per definition scheme corresponds to the SSA form, and register per access scheme allocates a new register in the basic block every time a live variable is defined in one of the predecessors of current basic block or read in the current block [1]. For the register per access strategy, a register is considered to be a live-in for the current block since the value is latched before the read during the execution of this block, and it is latched by the successor nodes in case of the write. So, compared to the register per definition, it adds registers for blocks where the variable is read.

In the case of a register per definition, a register is considered to be a live-out for the current block, which has a property that if there is a variable declaration in the basic block, the value of this variable is available in that basic block, instead of one of the successor nodes in the register per access scheme. Another property of this allocation scheme is that the number of basic blocks that have live-out registers

can be minimized, which reduces the number of inputs into the multiplexer to select a particular basic block scanchain. The datapath scanchain mechanism is described in Section 4.3, and allows the marshalling gadget to get and set the value of any live-out register in the SPE. Given that execution of blocks of code does not imply locality (i.e. it is not the case that a block of code that is added to a new version of the application will need only current basic block live-out values), the tradeoff of more registers for more locality increases the area of the SPE for a marginal benefit. Given the advantages of the reduction of number of registers in the SPE, and simplicity of the SSA form, a register per definition scheme is implemented in the current toolchain where registers are also placed after ϕ functions. The ϕ functions generate a new definition of the variable depending on the flow of control into the ϕ node. Thus, at any point of execution, there is only one last definition of any register and it is stateless since addition of registers after ϕ nodes eliminates the need to keep track of the control flow to find that definition

4.3 Datapath, control and gadget scanchains

The datapath registers are implemented as shift registers that have a regular input the width of the register, and a shift input and output of a variable width that have priority over regular input. All datapath registers in the SPE are connected by circular scanchains, where the number of scanchains placed in the design depends on the area and performance tradeoff. Increase in the bus width of shift wires decreases the time to shift the register contents in or out, while increasing the area of the design. During normal execution the registers receive data via conventional input, whereas, in the shift mode, the values of all registers on a given scanchain can be scanned out and scanned in by enabling scanchain shift. The circular connection of the scanchain allows scanchain rotation with the values of the registers preserved, while each of the register values can be set via normal input.

Similarly, the registers for the gadget and control paths are implemented as shift registers connected in a scanchain. These scanchains are not circular since the shift mode is only used during the initialization of the SPE to set the values

for the gadget registers and control path edge exception registers. Furthermore, the shift registers for gadgets and control path do not have a conventional input, since these values are set once during the SPE initialization. Thus, the current design employs three types of scanchains: datapath, gadget and control, where the datapath scanchain is used to transfer state between the SPE and the patch processor, and control and gadget scanchains provide a way to configure the SPE for a given application version.

4.4 Marshalling Gadget

The marshalling gadget serves as an interrupt dispatcher for the SPE. Upon encountering an exceptional condition (i.e., a patch that needs to be executed on the SPP), the control needs to be transferred to another processor along with the state of the SPE and an interrupt number, and the execution on the SPE needs to be stalled. Exceptional conditions for a particular version of the application are identified during patch generation and set during the initial SPE configuration. Once the exceptional condition is resolved, the marshalling gadget transfers the modified state to the SPE and resumes execution. The marshalling gadget is coupled with the external I/O interface that handles the communication between the SPEs and other processors in the Arsenal system.

The marshalling gadget employs scanchains as a primary way to transfer SPE state. While the marshalling gadget is designed to handle exceptional conditions, the semantics that it provides allow making function calls from the SPE, where the address of the function as well as the arguments are transferred to the processor.

4.4.1 Configurable control path

This section describes the modifications to the control path that enable transfer of control between SPEs and SPP.

In the baseline Arsenal architecture, the control path is represented as a finite state machine (FSM), where nodes correspond to basic blocks and edges

correspond to data and control conditions. In order to execute a patch on the SPP, the SPE has to interrupt its execution and transfer control to the patch processor. To enable patching at an arbitrary node, a bidirectional edge is added for each node in the control FSM to the marshalling gadget with an exceptional condition that enables transitions on these edges. Upon encountering an exception, the control is transferred to the marshalling gadget that gathers the state needed for execution of the patch, marshals this state over the interface to the SPP, waits for SPP completion signal, marshals the new state to the SPE and resumes execution by transitioning on the edge to next basic block.

The patching is done on the edges of the FSM. Edge patching has a property that if a basic block has been modified, only the incoming and outgoing edges of this block have to be patched. Moreover, blocks are executed atomically in a sense that we know upon executing state n and before the transition to node m that all operations in state n have completed and committed, and none of the operations state m have started. For instance, Figure 4.4 shows a control flow graph, where node Z has and edge to A, and node A has edges to node B and C.

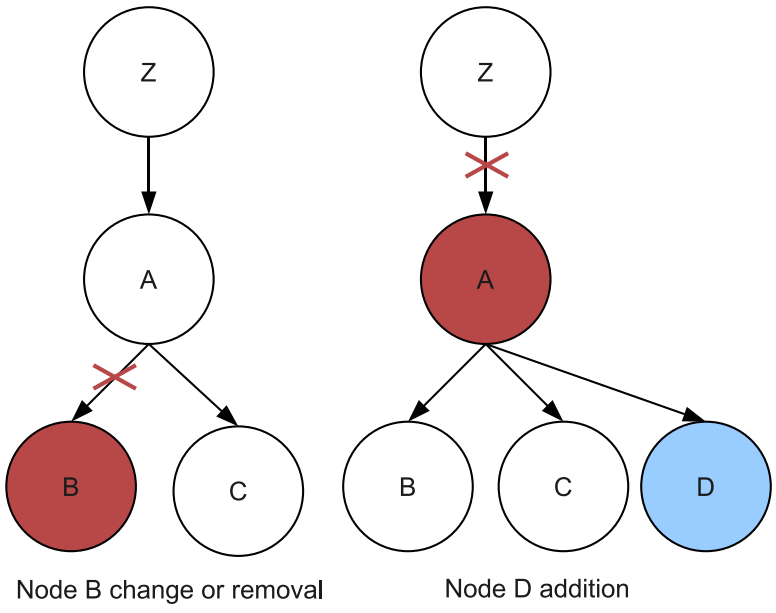


Figure 4.4: Modified edges in case of node change, addition or removal.

Lets first consider cases where corresponding application code for node B is changed in way that it cannot be executed on the original hardware, or if node B is removed in the new version of the application. Then, only edge AB is marked as exceptional condition, since there is no more edges incoming to B, and the outgoing edges will never be taken. If path A->C is taken, the execution proceeds in a normal way and there is no overhead of interrupt. If path A->B is taken, basic block A is executed completely and corresponding live registers are written. Then the execution is stalled and control is transferred to the marshalling gadget. The last case of node modification is a node addition, which implies the addition of a new edge as well. Lets consider an example where node A has two edges to node B and C, and a new block D with a corresponding edge AD is added. In order for this to happen, there has to have been a change in the logic of node A to include new edge, therefore, node A has been modified and the patching will be done on the edges incoming into A.

Thus, each state in the control path FSM includes 1-bit shift registers connected together in a scanchain that are placed for each transition from current control block and indicate if particular edge is to be taken. If this exception bit is set for some edge out of the control block, then, instead of transitioning to the next state, transition to marshalling gadget takes place. The initialization of the scanchain is done during the initial configuration of the SPE for a given application version and is described in Section 4.5.

4.4.2 State manipulation based on scanchains

The inputs and outputs of the datapath scanchains are statically routed from each SPE to the patch processor that are located within the same cluster, and the shift control on the datapath scanchains is done by the SPP. If the patch processor is located on another cluster, the datapath scanchains are connected to the SPE cluster arbiter. The SPE cluster arbiter, in turn, multiplexes all the input, output and shift enable wires to the SPEs in the cluster, and is configured by the patch processor to select the appropriate SPE for communication. Therefore, each datapath scanchain in the SPE has three connections to either the SPP or the SPE

cluster arbiter: input, output and shift enable, where the width of input and output shift wires is configurable. The number of scanchains per SPE is configurable as well and depends on the number of available connections on the patch processor side as well as the area and performance considerations. While smaller number of scanchains minimizes the number of interconnects between the SPEs and the SPP, the average time to access the contents of a register increases. Figure 4.5 shows the block diagram of two SPEs that contain two scanchains each connected to the shift controller on the patch processor that is located on the same cluster. The shift controller consists of a set of shift registers whose input, output and shift enable lines are connected to the datapath scanchains on the SPE. Through a set of instructions described in Section 4.4.3, the patch processor is able to select scanchains on one of the SPEs, rotate each scanchain, read and write values to it. Thus, this mechanism provides an abstraction to the patch processor that it is able to access any register in the SPE cluster through a set of shift registers allocated on the SPP.

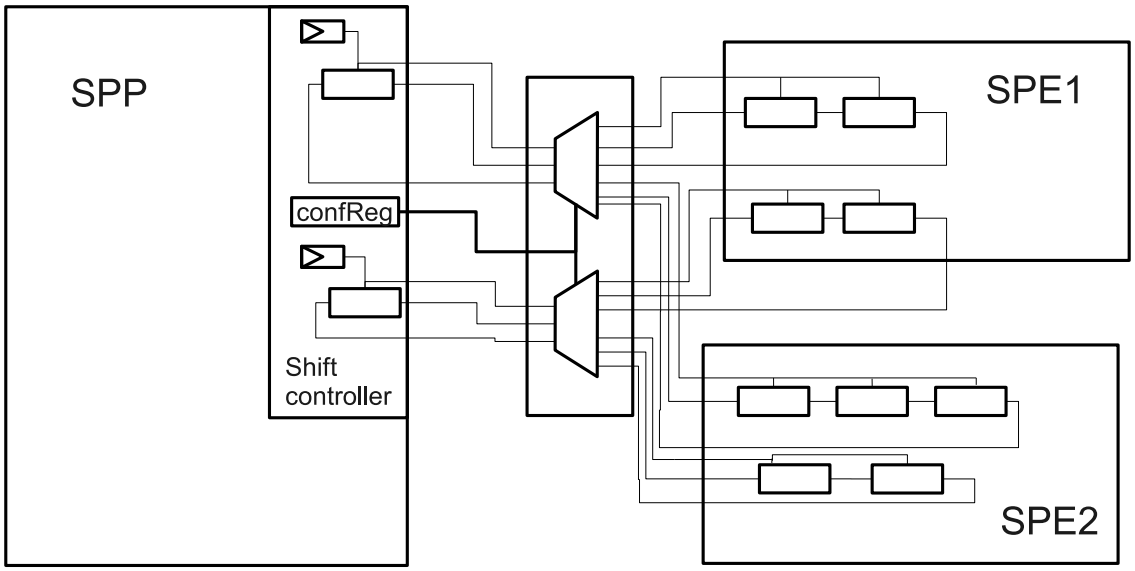


Figure 4.5: Block diagram of SPE cluster interconnect.

The registers that contain information relevant to the control state of the

SPE, such as, exception patch index in the interrupt table, edge number that caused the exception and the pointer to the local stack, are placed on the state scanchain that is connected to the state shift register in the SPP shift controller.

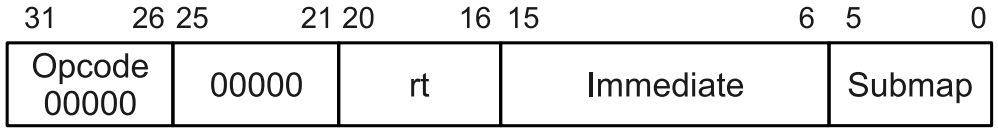
Upon encountering an exceptional condition in the SPE, the control is transferred to a marshalling gadget. The marshalling gadget asserts the SPP interrupt and transfers the interrupt vector index to the SPP. This interrupt number corresponds to the exception patch for a given version of the SPE, and is sent to the SPE by the runtime during the initial SPE configuration. The exception patch is described in Section 4.5. After the interrupt number transfer, the control is delegated to the patch processor and marshalling gadget waits until SPP signals completion. The interrupt handler on the SPP dispatches the appropriate exception patch based on the interrupt index. The patch processor then scans out the value of the offending control edge in the SPE by rotating the state scanchain and reading the value. Based on the value of this edge, the patch processor either starts executing if it does not require any state from the SPE, or starts reading required register contents from scanchains. Since the length of the scanchain and position of each register is known, the SPP initializes counters that are associated with shift registers in the shift controller to rotate the scanchains such that the required values are visible on the patch processor. The scanchain rotations are done in parallel, thus, if the patch needs register values from more than one scanchain, the SPP issues rotate instructions on these scanchains and blocking read instructions to get the values. The shift registers in the patch processor have data valid bits that are asserted once the associated counters reach zero, i.e. once the scanchain is rotated properly, which provides blocking register read and write semantics. Once the patch processor scans out the required state, it executes the code associated with a given edge exception. If the state of the SPE has been modified during the patch execution, the patch processor patches the modified registers via the same mechanism of scanchain rotations and register writes. The return state of the SPE is scanned back to the state scanchain, and, finally, the SPP asserts done signal. Upon receiving done signal, marshalling gadget sets the next state of the control path FSM to the value returned from SPP and resumes the execution on the SPE.

4.4.3 Specialized Patch Processor

An SPP is a general purpose processor that is customized to handle exceptional conditions in the SPEs due to changes in the software. The design goal of an SPP is a small, RISC style processor that is dedicated to a limited set of SPEs, thus providing reduced communication latencies compared to the general purpose processor, and a customized interface to SPEs for state information transfers. The current implementation of the SPP is an execution core on the RAW tile [15] that is allocated per cluster of the SPEs. Each RAW tile in the RAW tiled microprocessor design contains fetch unit,data and instruction caches for the execution core, switch processor with the Scalar Operand Network (SON) logic, and trusted and untrusted cores with the generalized transport networks logic. The execution core is similar to a conventional MIPS style pipeline with floating point unit and supports a classical MIPS ISA. The size of a raw tile in the 180 nm process is 16 mm^2, where an execution core without the FPU takes up approximately 15% of the area. Thus, a rough approximation of the size of this core is 2.4 mm^2 in the 180 nm, or 0.6 mm^2 in the 90 nm process. While this design provides an approximation for the requirements of the SPP and is present in the Arsenal toolchain, it is not area optimal and includes extra hardware that is specific to the RAW implementation. Given the fact that patch execution is an exceptional condition, with the nominal frequency of occurrence and the number of instructions to execute, the area and power requirements for the SPP outweigh the performance requirements. According to MIPS [11], an area-optimized implementation of the M4K core has a die area of 0.12 mm^2, typical power consumption of 0.04-0.15 mW/MHz, and a worst case maximum clock speed of 200-414 MHz, when implemented in a TSMC 90 nm process. M4K core is a classical 32 bit RISK core that supports full MIPS ISA, with the 5 stage pipeline, no on-chip data or instruction cache, integer ALU with multiply and divide units, MMU, and an SRAM interface. The area of this design is on the order of a small SPE, thus allowing a placement of a general purpose core in each one of the clusters. Since the L1 cache is present on the cluster, the SPP shares the data caches with the rest of the intra-cluster SPEs, while the instruction cache is not shared and located off-chip. The size of the instruction

cache was determined by the analysis of the number of instructions of the code fragments in Section 5.1, where the code fragment is defined to be a block of code that is either introduced or removed in the new version of the application. Thus, this is a measure of the maximum number of instructions in a contiguous block that an SPP will need to execute. Even though the maximum number of instructions in the CDF of block instruction sizes on Figure 5.5 exceeds 1024, blocks of code of this size are infrequent and indicate a drastic change to the underlying application. Thus, faced with the revamp of the algorithm, the software fallback to a powerful general purpose processor is a more likely scenario. The majority of the block instruction sizes fall within a range of 16 to 32 instructions, therefore the size of the instruction cache in the SPP is set to be 128 bytes with an area of approximately 0.018 mm^2 [16].

The ISA modifications of the RAW execution core reflect the architectural changes to accommodate requirements for the SPP. Four instructions were added to control the scanchain registers on the SPP side with instruction format shown on Figure 4.6, where the immediate field is used to reference scanchain register number, and *$rt* for a general purpose register.



Submap
110000 : mfsc $rt ← SPE[confReg].SC[Imm]
110001 : mtsc SPE[confReg].SC[Imm] ← $rt
110010 : scrl SPE[confReg].SC[Imm] rotated by $rt bits
110011 : signal Imm ? (SPE[confReg].activate←1) : (confReg←($rt<256 ? $rt:-1))

Figure 4.6: Scanchain instruction format.

The *signal* instruction either sets the configuration register to the value in *$rt* that holds the SPE number if *Immediate* is non-zero, or sends a signal to the SPE if immediate field is zero. The configuration register is a select on either the multiplexer for the data and shift enable signals if patch processor is located

on the same cluster, or on the SPE cluster arbiter, if patch processor is located on separate cluster. Move from scanchain instruction, *mfsc*, moves word from one of the scanchain registers in immediate field to a general purpose register *$rt*. Move to scanchain instruction, *mtsc*, moves word from a general purpose register *$rt* to a scanchain register number in immediate field. The scanchain rotate left instruction, *scrl*, rotates scanchain number set in immediate field by number of bits in *$rt*.

4.5 Patch generation and deployment

The Arsenal compiler generates the configuration patch that contains initialization information for each SPE, and an execution patch for the SPP that is invoked to handle exceptional conditions.

Since the architecture of the SPE is known, and the structure of the SPE corresponds to the structure of the code, generation of the patch is based on the comparison of basic blocks of the SPE and the control and data flow graphs. Given two versions of the application with graphs G and G' and an SPE that was built for graph G, two basic blocks B $\in$ G and B' $\in$ G' match if the registers and data flow dependencies of these blocks have a one to one correspondence. The basic block matching of set of constructs S that can be handled by gadgets (i.e. operators, constant values, etc) is delegated to gadget matcher that returns a match if this set is identical. If the set S differs, but can be patched by gadgets in the basic block B, the match is returned and a basic block configuration patch is generated for the basic blocks in question. This patch contains the initialization values for the gadgets. Finally, if the differences between basic blocks cannot be localized by gadgets, the blocks are marked as non matching. The control flow matching is done on the edges of the graph. Given an edge e = (u,v) $\in$ G, and edge e' =(u',v') $\in$ G', there is a match between e and e' if the source basic blocks u and u', and destination basic blocks v and v' match, and the edge label (u,v) matches (u',v'). The mismatches produced by this algorithm are incorporated into the execution patch, which handles the new parts of the code that cannot be executed on the

SPE. Since patching is done on the edges of the graph, the execution patch maps the edge transition to the new code that is to be executed. The data dependencies of the new code are mapped to the datapath basic block scanchains so that the appropriate scanchains can be both scanned out from the SPE and scanned in to the SPE if the SPP execution affected the state of some variables in the SPE.

Thus, the exception patch is invoked by the exceptional condition on the SPE. Upon receiving the edge ID from the SPE, the exception patch on the SPP knows which basic block triggered the exception, and what code needs to be executed. It acquires the state, if needed, from the SPE, executes the code, patches the affected state, if any, back to the SPE and sets the appropriate return state in the control graph of the SPE.

The configuration patch is invoked during the SPE configuration. The runtime dynamically maps SPEs to execute code that is found suitable for a given SPE. All SPEs contain a version number that indicates an application version that SPE is currently initialized to. When the SPE is allocated, the runtime checks its version number and, if the application and SPE versions do not match or the SPE is executed for the first time, configures the SPE with the corresponding version patch. This configuration patch comprises of the current version number and initialization data for control path and gadget scanchains. Since data path scanchain contains only runtime information, it is not included in the configuration patch. All subsequent runs of this application version do not require configuration, unless another version is executed in between. Thus, the cost of initial SPE configuration is amortized over all subsequent calls to this SPE. The patch deployment mechanism for each scanchain is implemented as a simple FSM that initializes the counter to the number of bits in the patch, sets the shift enable signal for the registers on the scanchain, and shifts in values until counter reaches 0. Both scanchains are initialized in parallel, however, the initialization time depends on the amount and complexity of the gadgets and on the number of edges in the control graph of the SPE. Thus, initialization time is the maximum of the times to initialize each scanchain. The layout of a configuration patch is shown in Figure 4.7, where offset indicates the offset of the gadget scanchain path, and size1, size2 indicate the size

in bits of control and gadget scanchains patch respectively.

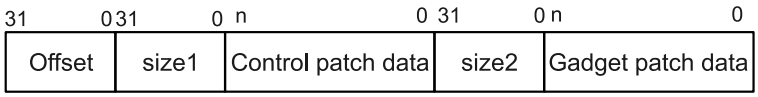


Figure 4.7: Configuration patch layout.

Runtime can dispatch SPE configuration in a blocking manner, where the runtime sends the configuration patch along with the rest of the arguments to the SPE, or in a non-blocking manner, where the runtime sends configuration patch to the SPE and continues execution, while SPE fetches patch information and initializes scanchains. However, non-blocking patch deployment requires either buffering on the SPE side, or reducing the amount of information that runtime needs to send to the SPE. The first approach can be implemented by allocating message queues for the SPEs that buffer the patch information. The second approach can be handled by communicating through a memory interface, where the runtime sends only the pointer to the configuration patch and resumes execution, and the SPE loads the patch via load units and feeds it into scanchains. The current version supports non-blocking patch deployment via memory interface, where the runtime sends the pointer to the configuration patch along with the rest of the arguments to the SPE.

4.6 Example of execution on a configurable ASIC

This section gives an example of function execution on a baseline SPE and on the configurable SPE that includes marshalling gadget, SPE configuration module and 3 types of gadgets: ALU for each operation, configurable data structure offsets, and configurable constants. The source of the sample function is listed below, where the code marked with "-" is the code removed in the new version, and code marked with "+" is the code added to the new version.

```
typedef struct {
        int num;
+       int err;            //change of the offset of foo.res
        int res;
}struct_baz;
int sample_function(int n, struct_baz *foo) {
        int i, sum;
-       sum = 0;
+       sum = 1;

        for ( i=1; i<=n; i++ ){
-               sum <<= i;
+               sum += i;     //operation change from << to +
        }
        foo->res = sum;
        foo->num = n;

+       if ( sum <= 0 ){              //block of code addition
+               fprintf(stdout, "Foobaz!\n");
+       }
      return sum;
}
```

Figure 4.8 shows a simplified block diagram of the baseline ASIC, where the control and data paths are shown separately. Dotted lines indicate basic block or cycle boundaries. This baseline ASIC is only capable to execute the original version of the sample function, and takes the software fallback approach to execute the new version.

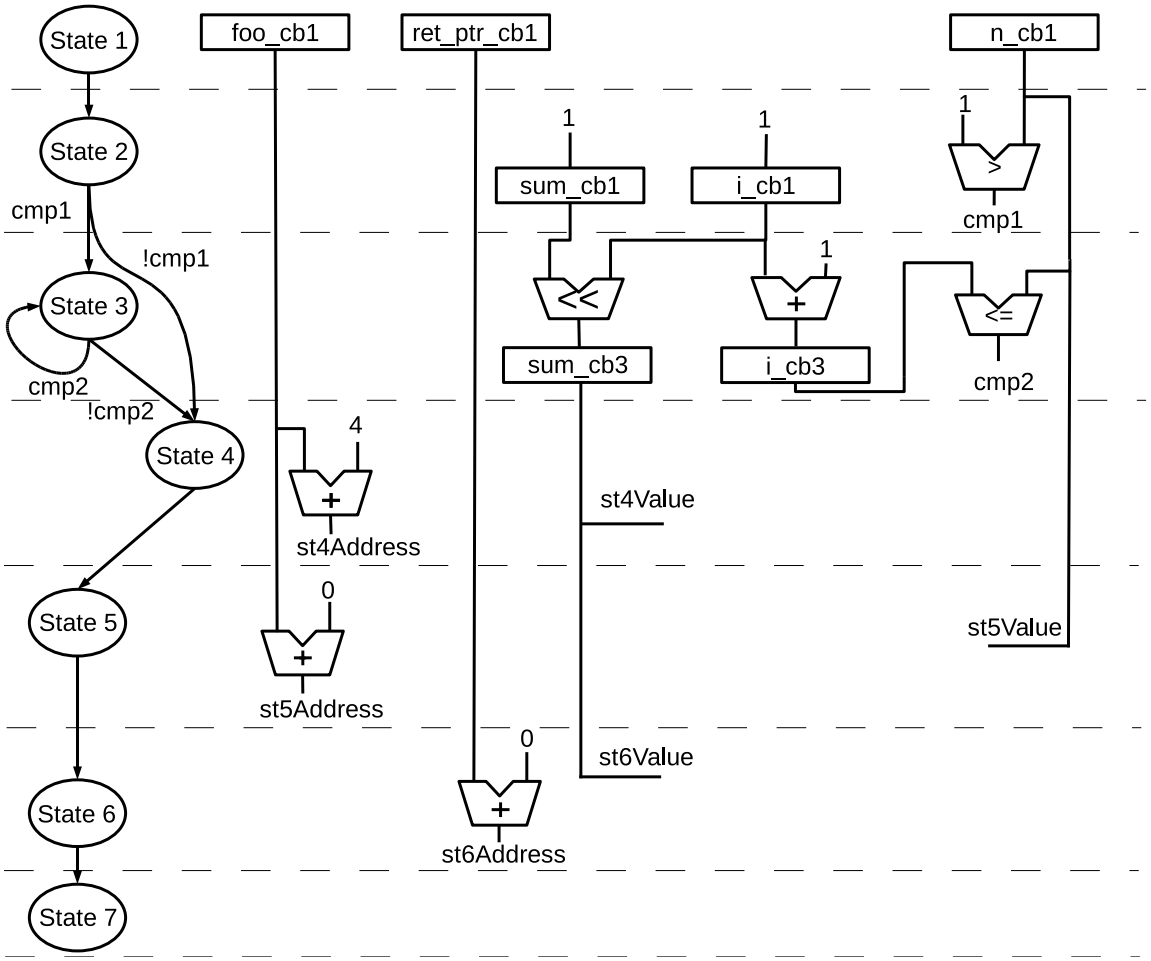


Figure 4.8: Block diagram of a baseline SPE.

The execution per basic block proceeds as follows:

state = 1. Arguments *int n* and *struct_baz *foo* are latched to *n_cb1* and *foo_cb1*, return pointer is saved in *ret_ptr_cb1*.

next_state =2

state = 2. Initial assignment:

i_cb2 = 1

sum_cb2 = 1

Loop bounds check:

cmp1 = (1 > n_cb1)

next_state = (cmp1 ? 4:3)

state = 3. Loop body:

sum_cb3 = sum_cb3 << i_cb3

i_cb3 = i_cb3 + 1

Loop bounds check:

cmp2 = (i_cb3 <= n_cb1)

next_state = (cmp2 ? 4:3)

state = 4. Store of sum_cb3. *//foo.res = sum;*

st4Address = foo_cb1 + 4

st4Value = sum_cb3

next_state = 5

state = 5. Store of n_cb1. *//foo.num = n;*

st5Address = foo_cb1 + 0

st5Value = n_cb3

next_state = 6

state = 6. Store of return value. *//return sum;*

st6Address = ret_ptr_cb1 + 0

st6Value = sum_cb3

next_state = 7

state = 7. Final state. Done signal is asserted.

New version of the function has several changes: the offset of *int res* in *struct foo* changes due to addition of member *int err*, initial value of *int sum* changes, operands change from "<<" to "+" in *sum <<= i;*, and a block of code is added that includes a conditional statement and a function call. The execution on the configurable version of the SPE is shown for the new version of the program, highlighting the points where execution paths differ. Figure 4.9 shows the control path FSM and datapath of the configurable SPE version, where the added hardware is drawn with dashed lines. This figure does not include the gadget and control path scanchains and SPE initialization hardware. The datapath registers in basic blocks 1, 2 and 3 are converted to shift registers and form three scanchains.

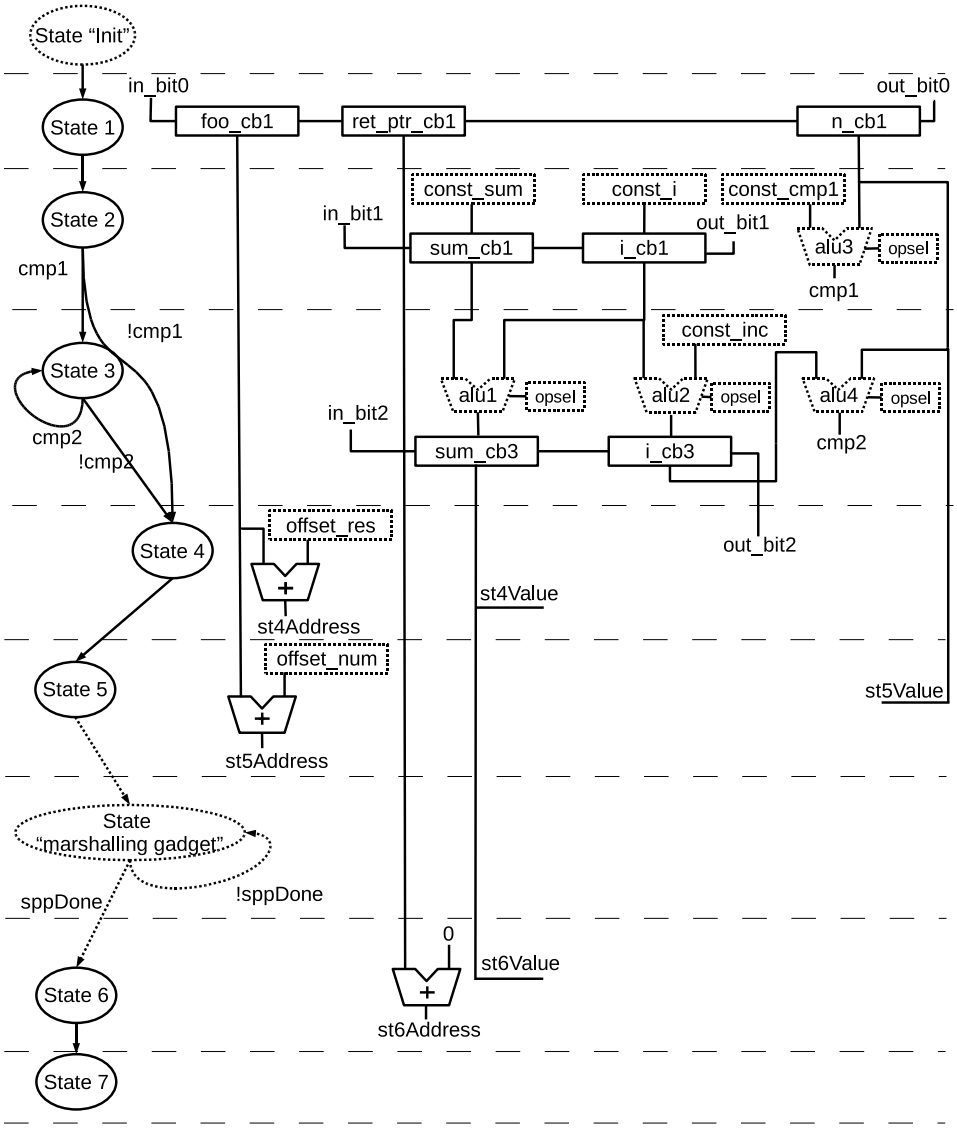


Figure 4.9: Block diagram of a configurable SPE.

The SPE first loads the configuration patch to set the appropriate values in the datapath gadgets and the exception bits on the edges of the control path. Since the addition of the code block cannot be handled by the SPE, the exception bit is set on the edge $e = (CB5, CB6)$, and the rest of the exception bits are set to zero. The datapath values to be loaded are shown in Table 4.2, where the values that changed between versions are marked with ”*”.

Table 4.2: Datapath values of the configuration patch.

Version 1	Version 2	Comments
const_i = 1	const_i = 1	32b initial value for *int i*
*const_sum = 0	const_sum = 1	32b initial value for *int sum*
const_cmp1 = 1	const_cmp1 = 1	32b constant in *1 > n_cb1* comparison
const_inc = 1	const_inc = 1	32b constant in *i_cb3 += 1*
*opsel_ALU1 = "<<"	opsel_ALU1 = "+"	4b ALU operation in *sum_cb3 = sum_cb3 (opsel_ALU1) i_cb3*
opsel_ALU2 = "+"	opsel_ALU2 = "+"	4b ALU operation in *i_cb3 = i_cb3 (opsel_ALU2) 1*
opsel_ALU3 = ">"	opsel_ALU3 = ">"	4b ALU operation in *cmp1 = 1 (opsel_ALU3) n_cb1*
opsel_ALU4 = "<="	opsel_ALU4 = "<="	4b ALU operation in *cmp2 = i_cb3 (opsel_ALU4) n_cb1*
*offset_res = 4	offset_res = 8	32b constant for *foo->res* offset
offset_num = 0	offset_num = 0	32b constant for *foo->num* offset

The execution per basic block proceeds as follows:

state = init. If the SPE executes a given version for the first time, the configuration patch is loaded and shifted to the gadget and control scanchains.

next_state = (initDone ? 1: init)

state = 1. Arguments *int n* and *struct_baz *foo* are latched to *n_cb1* and *foo_cb1*, return pointer is saved in *ret_ptr_cb1.*

next_state =2

state = 2. Initial assignment:

i_cb2 = const_i

sum_cb2 = const_sum

Loop bounds check:

cmp1 = (const_cmp1 (opsel_ALU3) n_cb1)

next_state = (cmp1 ? 4:3)

state = 3. Loop body:

sum_cb3 = sum_cb3 (opsel_ALU1) i_cb3

i_cb3 = i_cb3 (opsel_ALU2) const_inc

Loop bounds check:

cmp2 = (i_cb3 (opsel_ALU4) n_cb1)

next_state = (cmp2 ? 4:3)

state = 4. Store of sum_cb3. *//foo.res = sum;*

st4Address = foo_cb1 + offset_res

st4Value = sum_cb3

next_state = 5

state = 5. Store of n_cb1. *//foo.num = n;*

st5Address = foo_cb1 + offset_num

st5Value = n_cb3

After the execution of this basic block, the control is transferred to the marshalling gadget to handle the inserted block of code in the new version.

next_state = marshalling gadget

state = marshalling_gadget.

The marshalling gadget asserts the SPP interrupt, sets the exception edge number *e = (CB5, CB6)*, and transfers the interrupt handler pointer to the SPP. The SPP loads the exception patch, scans out the edge number that caused exception, and dispatches to the handler of that edge number. The state required by the exception patch is the value of variable *sum* located in *sum_cb3* register. The SPP rotates scanchain 2 that has *sum_cb3* register, reads the value, and executes the inserted block of code. Since none of the SPE registers were modified, the SPP does not need to transfer modified state back, and only needs to to resume execution on the SPE in appropriate state by setting *next_state = 6* on the control scanchain and rotating it. Once the patch processor finishes, it asserts the sppDone signal and SPE resumes execution.

next_state = (sppDone ? next_state : marshalling_gadget)

state = 6. Store of return value. *//return sum;*

st6Address = ret_ptr_cb1 + 0

st6Value = sum_cb3

next_state = 7

state = 7. Final state. Done signal is asserted.

Thus, given a code change between versions, the SPE is able to execute the new version by updating gadget configuration and setting appropriate edge exceptions for the changes that cannot be handled by gadgets.

5 Results

5.1 Identifying application changes between versions

The study of software evolution between subsequent releases aims to identify the frequently changing software constructs that are representative of the set applications targeted by the Arsenal architecture. The application set consisted of Lame MP3 Encoder, an MPEG Audio Layer III encoder, libjpeg, a JPEG encoder and decoder, and bzip2, a data compression/decompression project. We downloaded the latest releases that spanned several years, where libjpeg version set comprised of 11 versions (1-6b), Lame included 19 versions (3.88-3.92v2), and bzip2 included 6 versions (1.0.0-1.0.5).

All versions of the selected applications libjpeg, Lame and bzip2 were compiled with profiling and debugging support. Since instantiation of some application function in hardware requires an execution on the SPE to provide a performance benefit over executing this function natively, the functions that comprise the 95% of the total execution time were selected. We produced a list of differences per function between subsequent versions for all versions, where each change was manually classified. Each line of code was annotated with the static instruction count using PIN binary instrumentation tool [17] to estimate the footprint of the block of code that differs between versions on the instruction cache of the SPP.

Each change was classified along two dimensions, where the first one is an addition removal, or a change of the code in a newer version. All of these cases can cause an exceptional condition on the SPE if they cannot be handled by gadgets,

where the affected basic blocks are bypassed and the values for the blocks along the execution path are patched to the SPE. The second dimension targets the instruction level analysis of differences to identify the changes on the basic block level. Each affected statement was assigned to one or more categories: data flow, control flow, operator, interface and language changes, memory accesses, declarations, function calls, and code block addition and removal. These categories further subdivide into subcategories to target specific constructs. The language changes correspond to high level changes that are purely software constructs and do not affect the hardware instantiation. For instance, function scope changes, type casting and typedef declarations are examples of language changes. Interface changes account for addition and removal of the arguments to functions. Memory accesses are counted only for the first reference of data structure members and non-local variables between function calls. The declarations category includes only variable declarations without initialization. Operator addition, removal and change is placed in the operators category. Differences in loop constructs and conditional statements were broken down into control flow loops and control flow branches categories. The dataflow differences were classified into major and minor differences, where a major difference indicated that the statement was added, removed or changed substantially, whereas a minor difference indicated that the statement was modified while preserving the semantics and the meaning of the original statement. For instance, function calc_noise shows major dataflow addition and minor dataflow change between versions 3.96v1 and 3.97 of Lame.

Major: *+ noise = calc_noise_core_c(cod_info, &j, l, step);*
Minor: *- prev_noise->step[sfb] = step;*
+ prev_noise->step[sfb] = s;

The classification categories are not exclusive, thus, one change might be counted more than once if it is representative of several categories. Thus, an addition of a statement in function reverse_DCT between versions 4 and 4a of libjpeg

+ compptr = cinfo->cur_comp_info[ci];

was counted as 1 major dataflow, 2 memory accesses and 2 operator additions.

Addition or removal of several statements was classified as a block of code if it contained at least one control flow statement. The number of additions or removals of statements, function calls, loop constructs and conditional statements inside the block of code, as well as the number of static instructions per block of code was noted. Finally, if the function changed substantially between versions, it was classified as an overhaul and the changes in this function were not included in the total count.

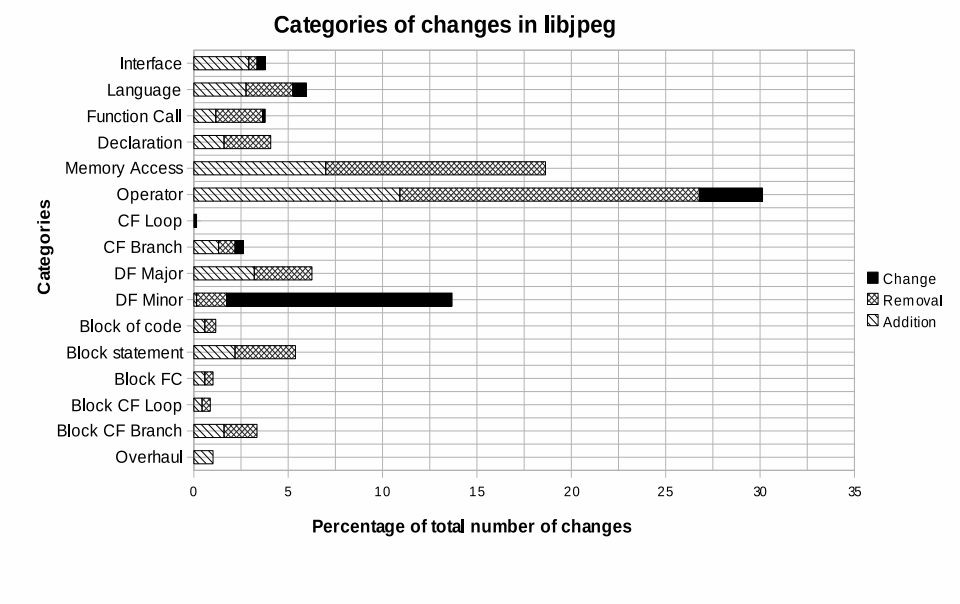


Figure 5.1: Categories of changes as a percentage of total changes in libjpeg.

Figures 5.1, 5.2 and 5.3 show the percentage of the total number of changes for each category of differences for libjpeg, Lame and bzip2 respectively. Figure 5.4 shows the cumulative number of changes per category for all applications in the test set.

These figures show that libjpeg and Lame have analogous results with the majority of changes being in the operators, minor dataflow and memory accesses. All of the categories shows similar percentage and ranking relative to each other. On the other hand, bzip2 shows an example of stable project that has only 44

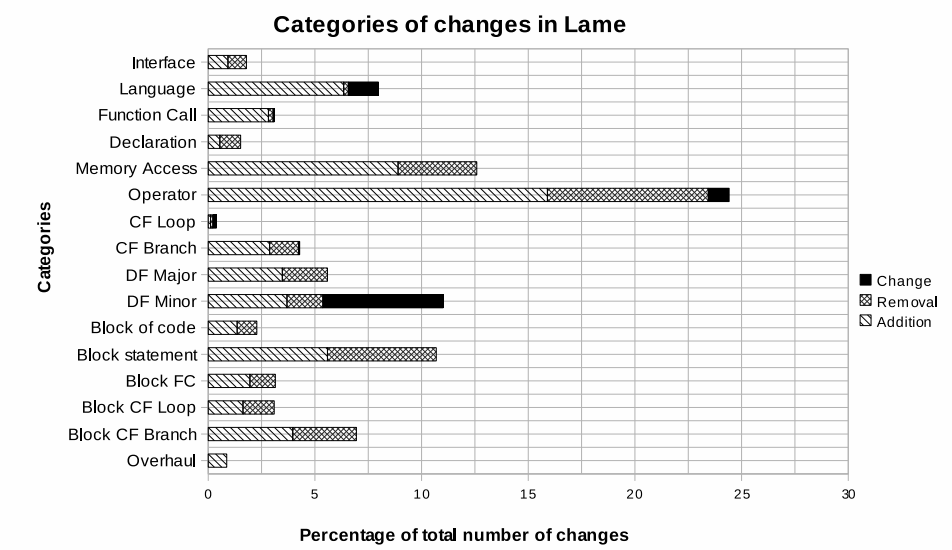


Figure 5.2: Categories of changes as a percentage of total changes in Lame.

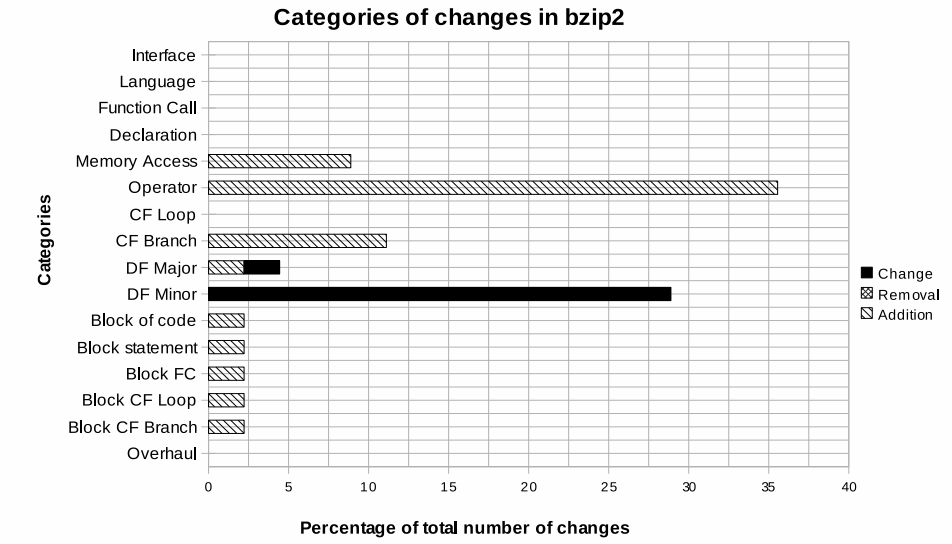


Figure 5.3: Categories of changes as a percentage of total changes in bzip2.

changes compared to 687 changes in libjpeg, and 1843 changes in Lame. Since we analyzed the versions of bzip2 after the 1.0 version, these results might not reflect the software evolution of a startup project.

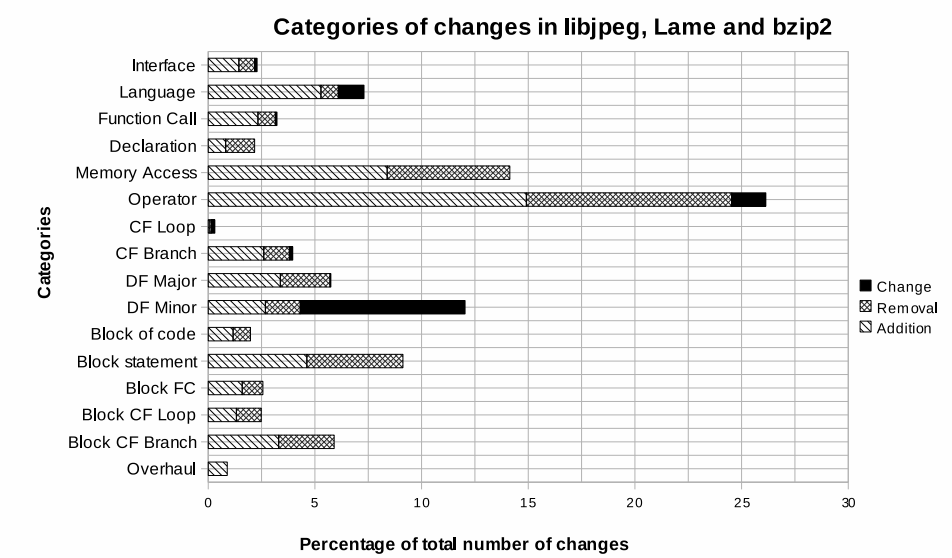


Figure 5.4: Cumulative number of changes per category for all applications in the test set.

Figure 5.5 shows the cumulative distribution function of the number of static instructions per code block or statement in libjpeg, Lame and bzip2.

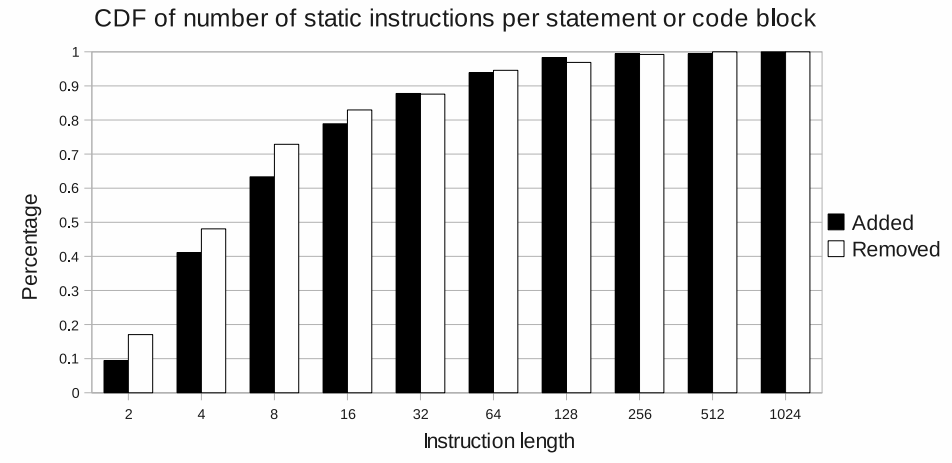


Figure 5.5: Cumulative distribution function of number of static instructions per added and removed blocks of code and statements.

Majority (87.7%) of the block lengths are less than 32 instructions, while

less than 1% of block lengths exceed 512 instructions. Given the infrequency of long blocks of code, and the fact that addition or removal of a sizable block indicates a major change to the function, the estimate of the footprint on the I-cache of the SPP was based on the instruction lengths below 32 instructions.

Another observation is that code addition is more common than code removal, which could suggest that code evolution involves more rewriting and addition of the new code, than old code elimination.

Figure 5.6 shows the CDF of the percentage of function versions with at least one change over the total number of versions that include a given function. Only 5.8% of the functions have changes in more than half of this function's versions, and 20.5% of the functions do not change at all.

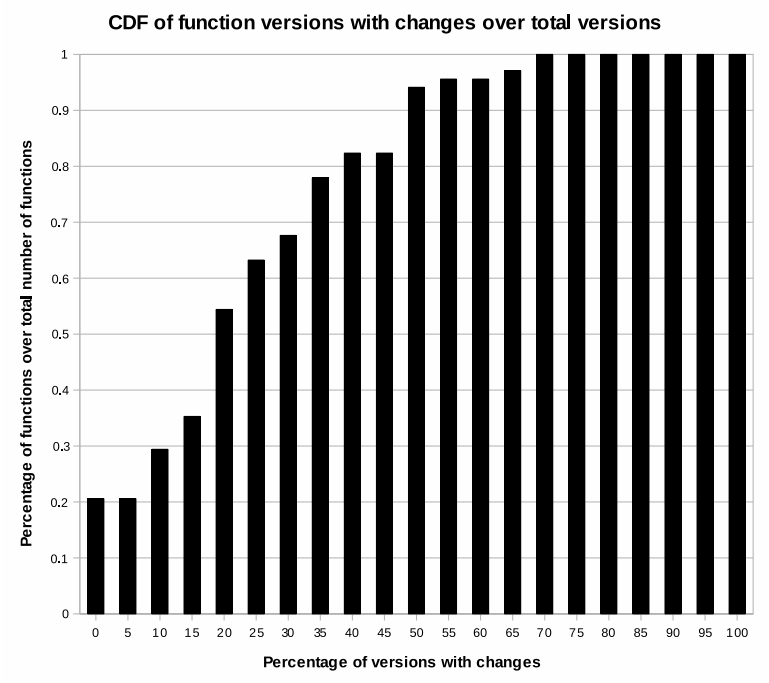


Figure 5.6: A percentage of function versions with changes.

The results show that majority of the changes are minor with the average length of added statements and blocks in the range of a few static instructions, which confirms the fact that the software evolution is mostly incremental, and the code changes are gradual. These incremental changes in the code evolution

reasserted our hypothesis that software is amenable to patching. However, there exists a subset of changes that is not practical to be handled by patching mechanisms. When faced with drastic changes in the function code, the ratio of the code that can be still executed on the SPE and the added code that is executed on the patch processor might be favorable to the software fallback mechanism. Out of a corpora of 756 versions of functions analyzed, 23 versions were overhauled, and out of 68 functions, 16 were changed substantially between subsequent versions at least once. Thus, on a function block level, the changes can be describes as minor and major. While patching mechanisms are applicable to the minor changes, the major changes represent structural or algorithmic shifts in the underlying application, where either the execution on the SPE is no longer feasible, or the SPE does not correspond semantically to the new functionality of the procedure.

This study is a preliminary investigation that was conducted on a limited set of an applications with the intent to identify the trends and paths of the software evolution, and an in-depth study is needed to validate the findings on a larger set of workloads and prove full generality of the results. However, this study gave us an indication that software patching is viable which served us as a basis for the exploration of the hardware patching capabilities.

5.2 Simulation results

This section discusses the methodologies and results we used to evaluate ASIC patching methodologies. All results are based on six configurations: *Baseline* (the non-patched version of the ASIC), *MG* (the patch enabled version that includes marshalling gadget and initialization logic to load configuration patch), *GadgetALU* (*MG* configuration plus configurable ALU gadget), *GadgetConst* (*MG* configuration plus configurable registers for constants gadget), *GadgetOffset* (*MG* configuration plus configurable registers for data structure offsets gadget), and *FullPatching* (MG configuration plus all of the gadgets). The *MG* configuration allows SPEs to handle software changes via the exception mechanism, whereas all subsequent configurations employ gadgets to mitigate effects of application changes

whenever it is possible with the ability to trigger exception upon a condition that cannot be handled by gadgets. To evaluate performance of the execution handling, we used the execution core of the RAW tile as a patching processor with 32KB L1 instruction cache, 32KB L1 data cache and simulation frequency of 950 Mhz. The width of datapath, gadget and control scanchains was set to one bit. We used 90nm CAD flow for design synthesis.

This work relies on the complex toolchain that currently lacks the maturity to provide cycle time data for full scale applications. In the interest of providing concrete results, the performance aspects of patching methods are shown on the *sample_function* SPE described in Section 4.6, and preliminary area and frequency results are shown for *scale_bitcount* SPE from Lame. Table 5.1 shows the synthesis results for the *sample_function* SPE that contains 4 ALU gadgets, 4 constant register gadgets and 2 data structure offset register gadgets.

Table 5.1: Area and frequency of *sample_function* SPE.

Configuration	Area, mm^2	Frequency, Mhz
Baseline	0.02646	288.34232
MG	0.06435	288.5753
GadgetALU	0.07599	247.91135
GadgetConst	0.06898	291.5962
GadgetOffset	0.06492	285.8613
FullPatching	0.08859	246.54832

The move from baseline configuration to the MG configuration increased the ASIC size 2.4 times. This increase is mainly due to the inclusion of the marshalling gadget and initialization modules that, when synthesized separately, occupy 0.026 mm^2 combined. Thus, there is a constant area increase that, for a full scale example, is less prominent. However, the non-constant part of area increase depends on the complexity of the SPE, namely, the number of registers to hold the edge exception bits, and the conversion of datapath registers to shift registers assigned to scanchains. Table 5.2 shows the simulation results for version 1 and 2 of the *sample_function SPE*. Since version 2 of the function has changes that utilize all of the gadgets and the block of code that requires execution on the patch processor,

the simulation results are shown only for the *fullPatching* configuration.

The cold start execution time includes the initialization of the SPE, while subsequent executions take the same number of cycles as the baseline SPE. Given the fact that the SPE only executes inside the loop body for 7 cycles, the configuration time dominates the runtime in this simple example. The execution of the second version of *sample_function* takes 41 cycles of SPE execution, 8 cycles to trigger the exception and transfer interrupt number to the patch processor, 288 cycles for SPE initialization on the first execution, and the rest of the time is taken to handle the exception on the patch processor.

Table 5.2: *sample_function* SPE simulation results.

	Version 1			Version 2		
Configuration	Total cycles	Initial exec. cycles	Subsequent exec. cycles	Total cycles	Initial exec. cycles	Subsequent exec. cycles
Baseline	82	41	41	N/A	N/A	N/A
MG	138	97	41	N/A	N/A	N/A
GadgetALU	167	126	41	N/A	N/A	N/A
GadgetConst	304	263	41	N/A	N/A	N/A
GadgetOffset	183	142	41	N/A	N/A	N/A
FullPatching	370	329	41	4981	3115	1866

Table 5.3 shows the area and frequency of the SPE for *scale_bitcount* function in version 3.94b of Lame. In version 3.96 two function calls are added in the beginning of the function that require execution on the patch processor:

+ assert(all_scalefactors_not_negative(scalefac, cod_info->sfbmax));

Thus, the baseline SPE becomes unusable after version 3.96, whereas the patchable SPE is able to execute eleven versions of this function. The *scale_bitcount* SPE has 43 configurable ALUs, 38 configurable constant registers and 8 configurable offset registers.

The frequency decreases between the baseline and different patching configurations range within 1.6% and 10.8%, and the area increase due to inclusion of patchable hardware ranges within 34.1% and 104.9% for *scale_bitcount* SPE.

Table 5.3: Area and frequency of *scale_bitcount* SPE.

Configuration	Area, mm^2	Frequency, Mhz
Baseline	0.28213	266.30449
MG	0.37842	261.96526
GadgetALU	0.52368	249.14047
GadgetConst	0.43002	256.10162
GadgetOffset	0.38778	260.62028
FullPatching	0.57833	240.20562

Currently, gadget substitutions are done for every construct to achieve a wide coverage of code changes, which results in a large number of gadgets that are not utilized in the patching capacity. This leads to modest frequency decreases and noticeable increases of the SPE area. Code analysis that identifies the constructs that are likely to change to be replaced with gadgets while leaving stable code sections intact can improve the performance of SPEs and preserve the benefits of configurable hardware.

6 Conclusion

This thesis presented the design of ASIC patching methods that extend the lifetime of the ASIC when faced with changes in the underlying application in the context of the Arsenal architecture. The resilience to software evolution is achieved by modification of the SPE with hardware gadgets that allow configurable runtime behavior, and mechanisms to transfer control and data between the SPE and the patch processor. If the application changes cannot be handled by gadgets, the flow of control is transferred to the patch processor that can access any register in the SPE complex via scanchain interfaces, execute the required code and resume execution on the SPE.

The results of study of software evolution on a set of sample applications show that software is amenable to patching, and that majority of changes in applications are localized, small and incremental. We demonstrated that differences between software versions can be abstracted in patches that allow SPEs synthesized from base application version to execute future versions of this application. The results show that hardware patching mechanisms can handle a wide range of changes in the underlying application code with reasonable performance overhead. However, our toolchain can currently provide simulation results only on simple applications. The gadget selection approach in this work targets the coverage of the diverse set of changes, thus exploring the generality aspect of the trade off between the SPE performance and generality of changes that can be handled. Further analysis of performance and coverage of various patching mechanisms for real life applications would provide insights into the appropriate gadget selection and generation that preserves the benefits of graceful degradation over several versions of the application while reducing the area, power and performance impacts

of inclusion of patching hardware.

References

[1] P. Briggs, K. D. Cooper, T. J. Harvey, L. Simpson. "SSA: Practical Improvements to the Construction and Destruction of Static Single Assignment Form". *In SOFTWARE PRACTICE AND EXPERIENCE*, vol. 28(8), 1-28 (July 1998)

[2] S. Narayanasamy, B. Carneal, B. Calder. "Patching Processor Design Errors". *Computer Design, 2006. ICCD 2006. International Conference on*. pp. 491-498, 1-4 Oct. 2007

[3] S. Sarangi, S. Narayanasamy, B. Carneal, A. Tiwari, B. Calder, J. Torrellas. "Patching Processor Design Errors with Programmable Hardware". *IEEE MICRO*, pp. 12-24, 2007

[4] S. Sarangi, A. Tiwari, and J. Torrellas. "Phoenix: Detecting and Recovering from Permanent Processor Design Bugs with Programmable Hardware". *Annual IEEE/ACM Intl Symp. Microarchitecture (Micro 06)*, pp. 26-37, 2006

[5] Glen Haas George Landers, Levon Petrosian, Neal Stollon, Les Veal. "RAMA: A Reconfigurable Datapath System-on-Chip Architecture". *Proceedings IC-SPAT*, 2000

[6] I. Neamtiu, M. Hicks, G. Stoyle, M. Oriol. "Practical Dynamic Software Updating for C". *In Proceedings of the ACM Conference on Programming Languages Design and Implementation (PLDI)*, pp. 7283, 2006

[7] LAME MP3 Encoder. http://lame.sourceforge.net/

[8] libjpeg Project. Independent JPEG Group. http://www.ijg.org/

[9] bzip-2 Project. http://www.bzip.org/

[10] Nathan Clark, Amir Hormati, Scott Mahlke. "VEAL: Virtualized Execution Accelerator for Loops". *ACM SIGARCH Computer Architecture News*, Vol. 36(3), pp. 389-400, 2008

[11] MIPS32 M4K Core Specifications. http://www.mips.com/products/processors/32-64-bit-cores/mips32-m4k/index.cfm#specifications

[12] M.B. Taylor, J. Psota, A. Saraf, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, A. Agarwal, W. Lee, J. Miller, D. Wentzlaff, I. Bratt, B. Greenwald, H. Hoffmann, P. Johnson, P. J. Kim. "Evaluation of the Raw microprocessor: an exposed-wire-delay architecture for ILP and streams". *ISCA 2004, Proceedings. 31st Annual International Symposium on Computer Architecture*, pp 2-13, 2004

[13] S. Swanson, A. Schwerin, M. Mercaldi, A. Petersen, A. Putnam, K. Michelson, M. Oskin, S. Eggers. "The WaveScalar architecture". *ACM Transactions on Computer Systems (TOCS)*, Vol. 25, Issue 2, May 2007.

[14] R Komondoor, S Horwitz. "Semantics-preserving procedure extraction". *Proceedings of the 27th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages (POPL)*, pp. 155-169, 2000.

[15] M. Taylor. "Tiled Microprocessors". *PhD Thesis, Massachusetts Institute of Technology*, February 2007.

[16] CACTI. http://quid.hpl.hp.com:9081/cacti/

[17] PIN, a Dynamic Binary Instrumentation Tool. http://www.pintool.org/