# Bundled Execution of Recurring Traces for Energy-Efficient General Purpose Processing

Shantanu Gupta* †, Shuguang Feng†, Amin Ansari†, Scott Mahlke†, and David August‡

†Advanced Computer Architecture Laboratory
University of Michigan
Ann Arbor, MI
{shangupt, shoe, ansary, mahlke}@umich.edu

‡ Department of Computer Science
Princeton University
Princeton, NJ
august@cs.princeton.edu

## ABSTRACT

Technology scaling has delivered on its promises of increasing device density on a single chip. However, the voltage scaling trend has failed to keep up, introducing tight power constraints on manufactured parts. In such a scenario, there is a need to incorporate energy-efficient processing resources that can enable more computation within the same power budget. Energy efficiency solutions in the past have typically relied on application specific hardware and accelerators. Unfortunately, these approaches do not extend to general purpose applications due to their irregular and diverse code base. Towards this end, we propose BERET, an energy-efficient co-processor that can be configured to benefit a wide range of applications. Our approach identifies recurring instruction sequences as phases of "temporal regularity" in a program's execution, and maps suitable ones to the BERET hardware, a three-stage pipeline with a bundled execution model. This judicious off-loading of program execution to a reduced-complexity hardware demonstrates significant savings on instruction fetch, decode and register file accesses energy. On average, BERET reduces energy consumption by a factor of 3-4X for the program regions selected across a range of general-purpose and media applications. The average energy savings for the entire application run was 35% over a single-issue in-order processor.

## Categories and Subject Descriptors

C.1.3 [**Processor Architectures**]: Adaptable Architectures

## General Terms

Design, Experimentation, Measurement

## Keywords

Energy Saving, Microarchitecture, Efficiency, Co-processor

---

*Author is currently with the Hybrid Parallel Computing Group at Intel Corporation

## 1. INTRODUCTION

The traditional microprocessor was designed with an objective of running general purpose programs at a good performance, while treating efficiency as a second order criteria. However, with a growing demand for on-chip resource integration, longer battery life and lower heat dissipation in modern day devices, there is an emerging need to improve computational energy efficiency. The trend in the silicon integration is also reinforcing this need for energy-efficient architectures. Over the years, transistor density and performance have continued to increase as per Moore's Law, however, the threshold voltage has not kept up with this trend. As a result, the per-transistor switching power has not witnessed the benefits of scaling, causing a steady rise in power density. Overall, this limits the number of resources that can be kept active on a die simultaneously [**?**]. An instance of this trend can be already seen in Intel's Nehalem generation of processors that boost the performance of one core, at the cost of slowing down/shutting off the rest of them.

While the importance of efficiency today is being felt across all domains of computing, from datacenters cooling costs to smartphone battery lives, a majority of past works have focussed on the embedded application domain. These solutions have leveraged specialized hardware units [16, 18, 21], loop accelerators (LAs) [6, 30], and wide-SIMD support [27, 9] to save energy. Unfortunately, these specialization approaches do not directly extend to general purpose applications such as desktop workloads, SPEC integer suite, OS utilities, library codes etc., for several reasons. First, these applications have a highly irregular program structure, and contain a large amount of control flow. For instance, the large, uncounted, non modulo-schedulable loops in these applications cannot be mapped to LAs [6, 4]. Second, these irregular codes exhibit little, if any, data level parallelism (DLP). This limits the applicability of SIMD support for energy savings. And finally, the general-purpose application space is very diverse and constantly evolving, Therefore, designing a custom hardware (like ASICs) for each of these programs is not very cost-effective.

Despite its shortcomings, specialized hardware like ASICs form an important design point (rightmost in Figure 1) in the space of techniques to improve performance and efficiency. In fact, a recent work [**?**] makes a case for function-level ASICs in the context of irregular codes, claiming the large availability of dark silicon. The advantage here is that carefully customized data-paths and long instruction ranges deliver highest levels of efficiency. The disadvantage of ASICs is that each of them can handle only *one* application/function. A second class of performance/efficiency solution that overcomes this challenge is the work on programmable functional units (PFU) [3, 19] (leftmost in Figure 1). PFUs allow a small (acyclic) chain of operations to execute together using a

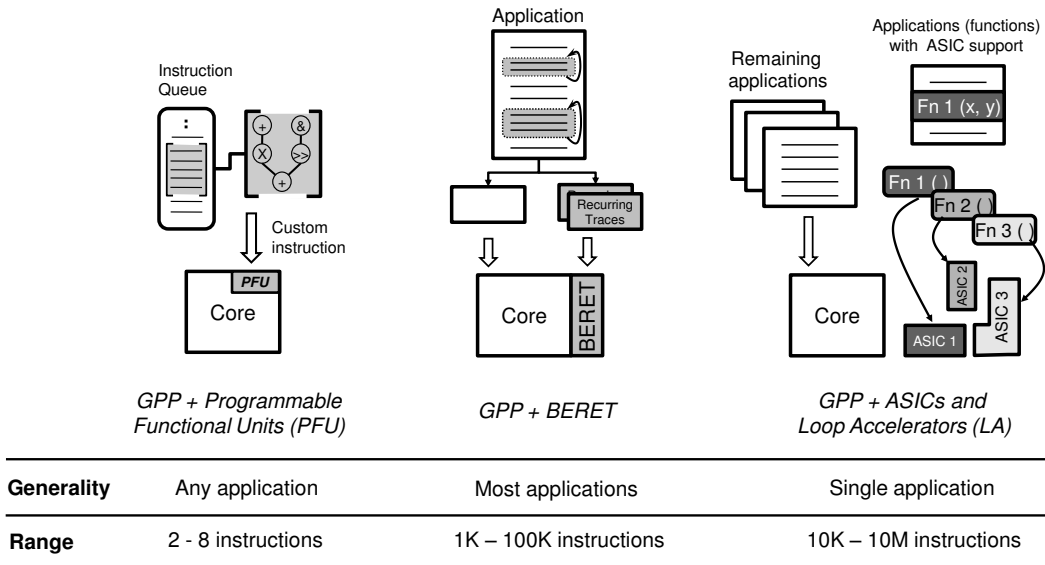| | GPP + Programmable Functional Units (PFU) | GPP + BERET | GPP + ASICs and Loop Accelerators (LA) |
|---|---|---|---|
| **Generality** | Any application | Most applications | Single application |
| **Range** | 2 - 8 instructions | 1K – 100K instructions | 10K – 10M instructions |

Figure 1: Solution classes to improve computational efficiency of general purpose processors (GPPs). Range of dynamic instructions off-loaded determines frequency of communication with the main core, and correlates well with energy savings. The PFU class covers custom instructions and subgraph accelerators that can target virtually any application, but work on small ranges. ASICs provide a much larger range, but are typically exclusive to an application's function. BERET is our proposed design point, which provides application flexibility while also covering large instruction ranges.

programmable/customizable set of functional units. The advantage is its universal applicability to almost any program. However, the energy efficiency gains are limited due to a small instruction range, and an emphasis on processor back-end. Studies have shown that a large fraction of application energy is consumed by the processor front-end [5].

The two solution classes (ASICs and PFUs) discussed above fall into opposing extremes, with one providing large efficiency gains, and the other a flexibility to work across applications. To bridge this gap, this paper proposes BERET (Bundled Execution of REcurring Traces), a configurable co-processor that achieves significant energy savings for the selected program regions mapped onto it, without sacrificing performance. As the approach tries to bridge the gap between PFUs and ASICs, it has to simultaneously achieve two objectives, increase instruction range relative to PFUs (which translates into better energy savings), and make the design flexible across applications unlike ASICs.

For increasing the instruction range, the insight is to leverage recurring instructions sequence in a program's execution. Such a sequence consists of instructions that repeatedly execute back-to-back with a high likelihood, despite the presence of intervening control instructions. These recurring sequences represent phases of "temporal regularity" in an otherwise irregular code, and make good candidates for mapping to BERET. Conceptually these are traces or frames [17, 8, 14], with an added requirement of forming a loop. Hereonwards, we refer to them as *hot traces* or *recurring traces*. The recurring traces provide several benefits such as long instruction ranges, predictable code behavior and appearance of structure to irregular codes, all of which help in designing a simple and efficient co-processor hardware. More importantly, as these traces are significantly shorter (15-20 instructions) than the original unstructured loops, BERET buffers them internally and eliminates redundant fetches and decodes.

The second objective of BERET is to support multiple applications. The insight here is to use a *bundled execution* model for running the traces. In this model, instead of executing one instruction at a time, BERET uses compiler analysis to break down traces into bundles of instructions, and executes them sequentially. These bundles are essentially subgraphs (chains of interconnected operations) from the trace-wide data flow graph. Further, our analysis of application traces demonstrated that many subgraph structures are common within as well as across applications. Thus, given a diverse enough collection of subgraph execution blocks, our compilation scheme is able to break down any given recurring trace into constituent subgraphs from this collection. In terms of energy savings, a major advantage of this bundled execution is that it significantly cuts down on the redundant register reads and writes for the temporary variables. Overall, we consider this bundled execution model a trade-off design that lets us achieve efficiency gains nearer to an application specific data flow hardware while maintaining application universality of regular Von Neumann execution model.

Leveraging these two insights, BERET is designed as a subgraph-level execution pipeline for recurring instruction sequences that enables significant energy savings for general purpose programs. Primarily, the energy savings come from large reductions in redundant fetches, decodes, and register reads and writes for temporary variables. BERET also represents a hybrid accelerator design point in the efficiency solution space where a large range of instructions are offloaded and most applications benefit.

## 2. A CASE FOR ENERGY EFFICIENT TRACE EXECUTION

In this section, we investigate the sources of inefficiency in a simple in-order RISC processor core, explore opportunities for energy savings, and detail our insights on designing a general purpose, energy-efficient compute engine. For a detailed comparison of our work to prior schemes, please refer to Section 6 and Table 1.
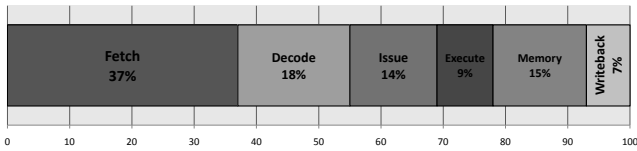
Figure 2: The distribution of energy dissipation across pipeline stages in an in-order processor.

## 2.1 Pipeline Energy Distribution

In a conventional Von Neumann architecture, the processor spends a large amount of effort in supplying instructions and data values to the actual execution units. The per-stage energy distribution in a simple in-order RISC processor (modeled after a MIPS core [5]) is shown in Figure 2. This data confirms that a large fraction of energy dissipation can be attributed to the instruction supply (Fetch and Decode). The major component behind this was the instruction cache, which is not only a large structure, but needs to be accessed for every single dynamic instruction in a program. The second biggest energy draw was from the combined register read (Issue) and write back (Writeback) cost. This is representative of the data supply cost, along with the datapath memory access (Memory). The last stage in this tally, surprisingly enough, is the data computation (Execute). Once the instructions and data are delivered to an execution unit, only a small amount of energy is required to compute the result.

This behavior clearly highlights that a regular in-order pipeline has a severe imbalance in terms of where the energy is being spent. For a small fraction of compute energy, almost 8X more energy is taken up to deliver the instruction and data to the execute stage. On a positive note, this also indicates that methods targeting instruction and data supply energy can achieve substantial savings.

## 2.2 Opportunities for Energy Saving

A significant source of this biased energy consumption is the lack of understanding a general purpose processor has for the underlying program structure. The hardware is typically agnostic of the presence of loops, live data values, data flow between instructions, chains of frequently occurring operations, and so on. This results in wasted effort for redundant instruction fetches and decodes (for repeating sequences such as loops), redundant register file reads and writes (for temporary / intermediate values), redundant forwarding and dependency checks for unrelated instructions, etc. Each of these redundant actions present an opportunity for energy savings.

A popular approach for reducing this wasted effort has been to introduce hardware specialization, in the form of ASICs [16, 18, 21], loop accelerators [6, 30], wide-SIMD support, etc. The attempt here is to encode the program structure in the hardware, such that it can avoid wasted effort during execution. For instance, loop accelerators buffer the instructions in a loop, thereby avoiding the redundant instruction cache accesses [6]. The hardware specialization solutions work particularly well for applications that have regular structure, data parallel computations, and limited control flow. Prime examples of this are multimedia and signal processing kernels.

## 2.3 Limitations for Irregular Codes

In addition to regular codes, energy-efficiency is equally important for applications in desktop computing, SPEC integer suite, OS utilities, libraries, etc. Unfortunately, the concept of hardware specialization, does not scale to this application class because:
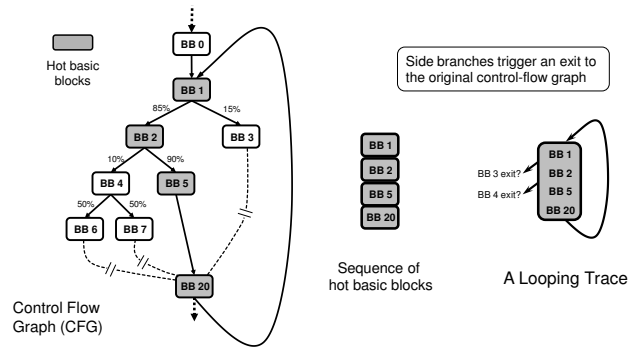


Figure 3: Extracting a looping trace from an irregular control flow graph. We also refer to these as *hot traces* or *recurring traces* in this paper, and use them as a construct that runs on our energy-efficient hardware design.

1. **Large and irregular loops:** The programs are highly irregular and contain a large amount of control flow. More specifically, the loops are usually large, uncounted (while loops), and contain deeply nested if-then-else statements. These characteristics are unfavorable for specialized hardware because: a) the hardware will be very large due to code divergence / loop size, b) it will have a low utilization because only single execution path would be taken each iteration, c) hardwired or simplified instruction delivery is lost due to the lack of code structure.

2. **Too many applications that are also regularly modified:** Even if one could somehow design ASICs for these *irregular codes*, a large number of such ASICs will be required to keep up with the application diversity and code modifications. This is unlike some embedded systems that have a limited number of relatively stable, well structured kernels.

## 2.4 Energy Efficiency for Irregular Codes

Due to the aforementioned reasons, achieving general purpose, energy-efficiency for irregular codes has long remained a tough challenge. In this work, we build upon two insights for solving this problem:

1. **Structuring the Irregular Code using Traces:** Often times, the dynamic behavior of irregular codes exhibits a regular structure. In the literature, this "temporal regularity" has been referred to as *traces* [8], *frames* [17] and *superblocks* [14] (in compilers). Traces are defined as sequences of instructions that have a high likelihood of executing back to back, despite the presence of intervening control instructions. These can be identified both statically and dynamically, covering roughly 70% of dynamic instructions [17].

   In the scope of this work, we focus on a subset of traces that also loop around with a high probability (hot or recurring traces). Figure 3 shows an example of an irregular control flow graph (CFG), with the extracted hot trace. These hot traces not only render a regular structure, but in addition, their looping nature is favorable to instruction supply energy savings.

2. **Generalizing Across Applications:** Working with hot traces eliminates the differences due to control flow between application codes, leaving behind only data flow variations. Further, we observed that hot traces can be segmented into small
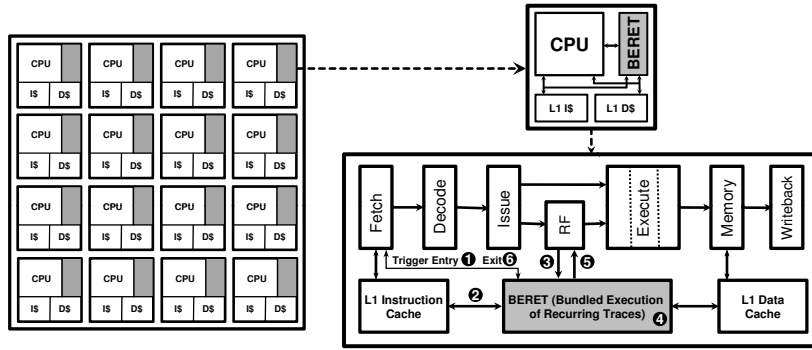
Figure 4: Deployment of BERET at multicore level and its integration within a single processor core.

data flow subgraphs, many of which are common across applications. Consequently, as we demonstrate later, given a diverse enough collection of subgraph execution units, a compilation scheme can be formulated to break down a trace into constituent subgraphs from this collection. The use of subgraph-based computation is also favorable for data supply energy savings.

## 3. THE BERET DESIGN

### 3.1 Overview

The proposed design, named BERET, is a configurable co-processor optimized for energy-efficient execution of hot traces from a program flow. These hot traces are short, logically *atomic*, single-entry, single-exit program regions with a high probability to loop back. Further, the BERET hardware executes these traces in bundles of instructions rather than individual instructions. One can think of these instruction bundles as data flow subgraphs from the trace. As a result of these high level design choices, several avenues of energy savings follow. First, the short program traces are stored inside BERET hardware, this eliminates redundant instruction fetches as traces loop around. Second, the instruction bundles from traces are encoded as BERET microcode, eliminating the need for decode. Third, use of instruction bundles helps in reducing unnecessary storage and retrieval of temporary values. And finally, the simplified design due to no branch prediction, smaller storage structures, and fewer pipeline latches, also contributes towards energy savings.

Conceptually, every core in a system can be augmented with an instance of BERET execution engine. Figure 4 shows this setup, and integration of BERET within a pipeline while sharing the same cache hierarchy. During a program's execution, whenever a (statically marked) hot trace is encountered, the fetch stage transfers control to the BERET hardware (Step 1 in Figure 4). BERET loads the configuration corresponding to this trace from the instruction cache (Step 2), and copies register live-ins for this region of code from the main pipeline (Step 3). At this point, the execution control has successfully transferred to BERET and it acts as an independent entity (Step 4). Internally, BERET executes the hot trace at the granularity of data-flow subgraphs, and repeats the sequence until a trace exit is flagged. More discussion about the BERET microarchitecture, its working, and challenges for trace exits, follow in Section 3.2. Once a trace exit is identified, the live-outs from this execution are copied back to the main pipeline register file (Step 5). And finally, a trigger is sent to the pipeline fetch stage, to start the regular program execution (Step 6).

Utilizing the BERET hardware involves identifying hot traces in a program's execution, and appropriately mapping them to the underlying BERET execution engine. Figure 5 shows a high level view of this process. The first step is to identify hot traces (Figure 5(a)) from the program execution that are good candidates for using BERET. The selected traces are frequently occurring sequence of program instructions that loop around, and rarely take a side exit. For every such hot trace, the instruction sequence is broken down into data flow subgraphs (Figure 5(b,c)). The subgraphs, if desired, can span across control instructions within a trace. In fact, the larger window of instructions visible in a trace supports this notion, and helps in identifying longer chains of connected operations. Finally, these subgraphs are mapped onto a heterogeneous set of subgraph execution blocks (SEBs) within the BERET hardware (Figure 5(d)).

In the above discussion, the latter few steps of dividing up a trace into subgraphs and mapping them to SEBs are interdependent, and thus, need to be handled concurrently. Section 3.3 details our compiler analysis and mapping algorithms for a near-optimal breakdown of traces into subgraphs supported by the BERET hardware. In order to decide this set of SEBs, we performed detailed analysis on traces from SPEC integer benchmarks, Linux utilities, encryption and media kernels. Section 4 discusses this procedure and also uses trace analysis to guide sizing of various microarchitectural sub-components within BERET (internal register file, configuration RAM, etc.).

The above described execution model of the BERET microarchitecture is quite effective at saving energy. These savings can be broadly attributed to: 1) reducing instruction fetch / decode cost and 2) reducing register access cost. First, once BERET buffers a trace for execution, there is no further instruction cache access. This eliminates redundant activations of instruction cache, fetch stage logic, and decode logic for the repeated sequence of instructions within a trace. Second, the register file accesses are cheaper as well as less frequent in the BERET design. The small size of the BERET's internal register file makes the accesses cheaper, while the subgraph execution model minimizes register reads and writes for intermediate variables in a program data flow.

### 3.2 Hardware Design

Unlike a regular pipeline, the BERET hardware deals with the execution of a small snippet of code ($\sim$20 instructions), containing a small number of a live registers ($\sim$6), and no internal control divergence. Further, the execution is conducted at the granularity of data-flow subgraphs, instead of individual instructions. These differences guide the following discussion about the design and working of BERET.
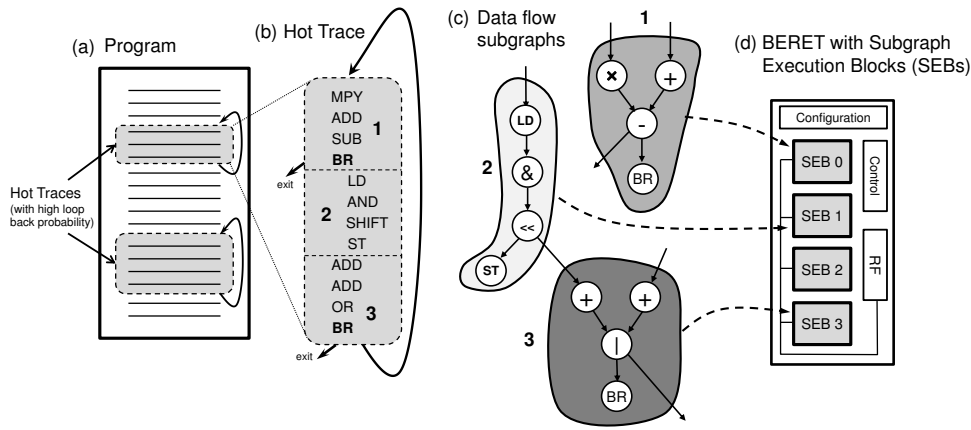
Figure 5: The process of mapping hot traces in a program to the BERET hardware: (a) shows a program segment with two hot traces, (b) a closer look at a trace with instructions and two side exits, (c) illustrates the break-up of trace code into data flow subgraphs, and (d) mapping of subgraphs to subgraph execution blocks (SEBs) inside the BERET hardware.

### 3.2.1  Basic Microarchitecture

Once a hot trace is configured to run on BERET, it acts as an independent execution engine. Given the small size of a trace, and no internal control flow, the BERET microarchitecture has a simplified front-end. However, it allocates significantly more resources to the execution back-end for running a wide variety of data-flow subgraphs. Figure 6(a) shows a block diagram of the BERET microarchitecture. Here, the configuration RAM (CRAM) stores the microcode for subgraphs in a trace, register file is for the internal data state of BERET, subgraph execution blocks (SEBs) are the equivalent of functional units, and control logic is to orchestrate the operation. In reality, the control logic is distributed across the entire fabric, with connections to virtually every component. The block diagram hides these connections for the sake of clarity.

Logically, BERET execution can be divided into three stages: 1) Configure SEB, 2) Execute SEB, and 3) Writeback results (Figure 6(b)). For every subgraph in the trace, the first step is sending (microcode) configuration bits to the mapped SEB. During this configuration stage, the register file inputs are also read into the input latch of an SEB. In the second stage (execute SEB), the SEB that has its inputs latched, configuration defined, and is in possession of the execution token, fires its functional units. The execution can take multiple cycles depending upon the subgraph depth (longest chain of instruction dependencies). Once the execution completes, the SEB sends the result on the writeback bus, and broadcasts an execution token. This token is now taken up by some other ready-to-execute SEB, and the pipelined execution continues. A more detailed stage-by-stage description follows below:

**1. Configure SEB:** The task of this stage is to sequence through the subgraphs in a trace, and configure SEBs to execute them. The configuration for the entire trace is stored on the CRAM. For each subgraph, this contains the SEB mapping, the register live-ins and live-outs, literal inputs, and mode bits for functional units within the SEB. In the first cycle, configuration bits are sent to the corresponding SEB, and register file access is made for two live-in values. In the second (optional) cycle, two more register live-ins can be read, or, when needed, the values are bypassed from the last executed subgraph.

**2. Execute SEB:** The second stage is responsible for the actual data computation on the SEBs. A SEB starts its execution when all the inputs are latched, configuration bits are available, and it possesses the execution token. The execution token is used as a serializing method to enforce in-order execution of subgraphs, and it keeps shuttling between SEBs. The execution can take multiple cycles, depending upon the subgraph depth, and concludes with values recorded in the output latch. In the event of cache miss, just like a regular pipeline, the SEB also stalls while waiting for the value.

Each SEB or subgraph execution block (Figure 6(c)) is an interconnected set of functional units (ALU, shifter, multipliers, etc), that represent a data-flow pattern. The number of functional units per SEB vary from two to six in our design space exploration (Section 4). The SEB structure has an input latch for live-ins, an output latch for live-outs, and a latch to store configuration bits. For every subgraph mapped, these bits decide the active functional units, their modes (add, subtract, etc), and flow of values between them. The selection of a good set of SEBs is central to the efficiency gains from mapping traces to BERET, and the pertaining discussion is presented in Section 4.

**3. Writeback:** This third and final stage is responsible for writing back the results from the last concluded subgraph execution to the BERET register file. All SEBs share a common writeback bus for this purpose, and any SEB that has its outputs ready, can request it. Due to the enforcement of in-order subgraph execution, there can never be a contention for this bus.

### 3.2.2  Handling Trace Exits

The microarchitectural description in the previous section assumes a straightforward execution scenario with indefinitely looping traces. However, in reality, the trace conditions would eventually dictate an exit, and a consistent program state has to be transferred back to the main processor. This is even more challenging when a side exit is taken in the middle of trace execution, because 1) the subgraphs are formed across control divergence boundaries, and assume that all instructions in the trace window execute in every iteration; 2) temporary register variables are excluded when mapping a trace to BERET, hence some of the live-ins required on the exit edge might not even be available. Figure 7 shows an illustration of control transfer between the main core and the BERET engine, copy-in and copy-out of values, and use of dual register file to maintain a clean state for last completed iteration.
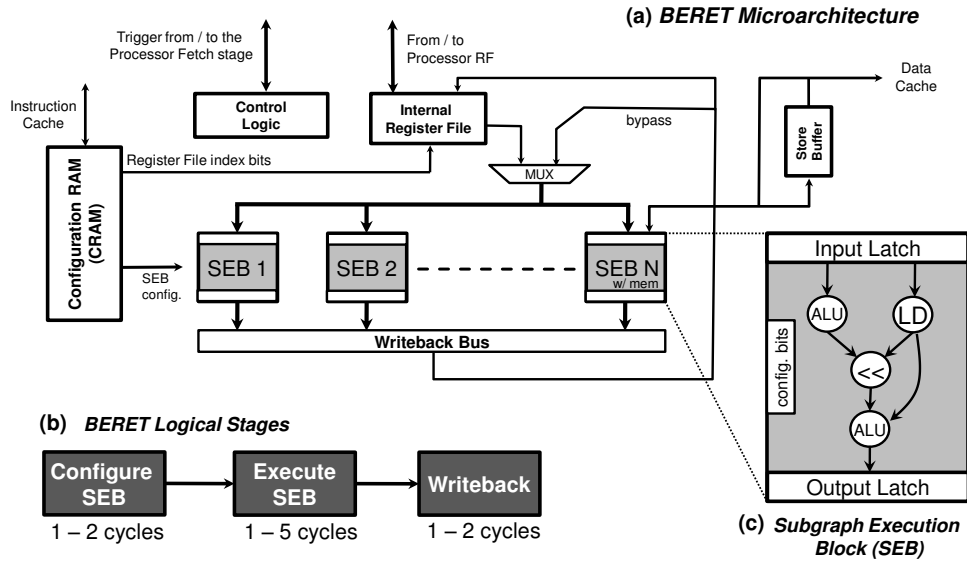
Figure 6: The BERET Microarchitecture: (a) the block diagram of the BERET hardware, (b) logical stages in the microarchitecture, and (c) a closer look at a subgraph execution block (SEB).

There are two parts to resolving this challenge. First, BERET needs a mechanism to detect when a side exit is taken by a trace. Second, BERET is required to maintain a committed state (at iteration boundaries) as well as per iteration speculative state. In the case of a side exit, the committed state (from last trace iteration) is copied back to main processor, which resumes execution from the trace head.

**Detecting Side Exits:** We first convert all the branches in the trace with assert operations (similar to [17]). The functional units within SEBs recognize this operation, and raise an exit flag whenever an assert computes to a true condition. Whenever any of the SEBs flag an exit, the control logic initiates the copying out of the committed state.

**Maintain Speculative and Committed State:** For recovering from early trace exits, BERET needs to maintain a committed state from the last completed trace iteration. There are two parts of this state maintenance: register file state and memory state. For register files, BERET uses a design similar to the concept of shadow register files. Essentially, every logical register maintains two physical versions in the register file. The even iterations of the trace write back to version 0 of registers, and odd iterations write back to version 1. As the code is linear within a trace, every iteration will produce exactly the same set of live-outs. Thus, at any point in time, the committed register state from the last iteration is available for recovery.

To maintain the memory state from the end of previous iteration, BERET buffers the stores from the current iteration. The store buffer releases them when the current iteration successfully completes.

### 3.2.3 Processor Interfacing

The main processor requires two modifications to interface with BERET. First, the fetch stage maintains a table of trace header addresses in the loaded program. Whenever the program counter hits any of these locations, the fetch sends an *entry* trigger and the corresponding trace configuration address to BERET. The BERET hardware loads the configuration using the instruction cache, runs

through the trace, and returns with an *exit* trigger to the fetch stage. The second modification allows the main processor's register file to be directly addressed by the BERET. This is required by the BERET to read the trace live-ins (at entry) into its internal register file and write back the trace live-outs (at exit).

## 3.3 Mapping Traces to BERET

We use a comprehensive compiler flow to identify and map hot traces from a program onto the BERET hardware. While mapping, the objective is to segment an identified hot trace into a minimum number of subgraphs, each of which can execute on a SEB in the BERET hardware. This compiler flow can be broken down into five major steps, each of which is elaborated below.

**Find Traces:** Given a procedure, the objective of this step is to identify traces that have a very high probability to loop back, and rarely take side exits. For this step, we leverage the previously proposed Superblock identification heuristic [14]. Superblock formation is a static compiler analysis that groups together program basic blocks with a high likelihood of executing one after another. This gives the compiler opportunity to perform optimizations on a larger window of instructions. From the set of Superblocks composed in a procedure, for BERET mapping, we select the ones that have a looped structure (branch from the last basic block to the first basic block), with an 80% probability to loop back.

**Enumerate:** In this step, all data-flow subgraphs are enumerated from the given trace. The subgraphs can range in size from one operation, all the way to a pattern of 4-6 interconnected operations. Since we are enumerating all subgraphs, an operation can appear in more than one subgraph.

**Map:** This steps checks the feasibility of which data-flow subgraphs can actually run on the BERET hardware, and prunes away the rest. The mapping phase iterates over the enumerated subgraphs, and attempts to map each of them to a SEB in BERET. If the subgraphs can map to multiple SEBs, the mapping to the smallest SEB is recorded. On the other hand, if the subgraph does not map to any SEB, it is discarded. In order to maintain mem-
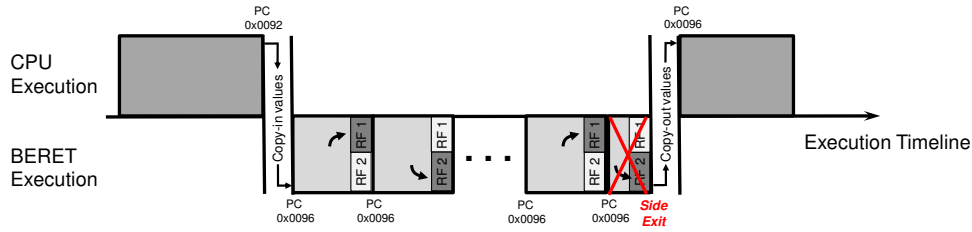
Figure 7: Execution transfer between the primary core and the BERET engine. The trace execution is terminated on BERET when a side exit is encountered, and register updates (from the last successful iteration) are transferred back to the main pipeline.
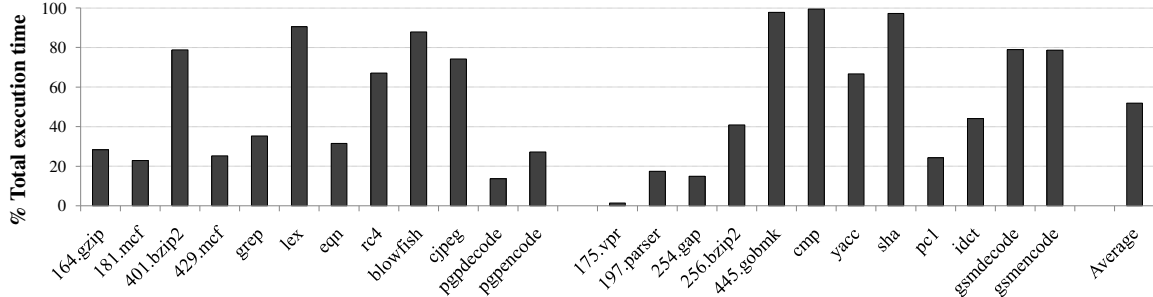


Figure 8: Fraction of execution time spent in hot traces for all 24 benchmarks. The average coverage is 54%.

ory consistency, a subgraph is also discarded if it contains memory instructions that alias.

**Select:** The input to this step is a set of SEB executable data-flow subgraphs from a hot trace. The selection phase is responsible for choosing the smallest subset of these subgraphs, while covering all instructions in the trace. This is equivalent to the set covering problem, which is NP-hard. We model it as a unate covering problem, and solve it using a branch and bound heuristic.

**Group:** Many of the subgraphs under utilize the SEB where they are mapped. This phase coalesces disconnected subgraphs, wherever it is possible, and places them on a single SEB.

After these steps for mapping, the compiler generates a configuration RAM code for this hot trace, and embeds it into the program binary. For ISA compatibility reasons, this configuration can be added as a part of the global data segment. Note that the compiler does not replace the original set of basic blocks in the program, as the execution reverts back to them in cases of early trace exit. Further, this keeps the code compatible on machines that do not have the BERET hardware.

## 4. DESIGN SPACE EXPLORATION

The previous sections assume a fixed hardware design for BERET, including the set of SEBs and sizes for different structures within the microarchitecture. This section explains our methodology to arrive at these design specifics. All experiments here were conducted on traces from SPEC integer benchmarks, Linux utilities, encryption kernels, and media kernels.

### 4.1 Benchmark Sets

We start with a total of 24 benchmarks from SPEC 2000 and 2006 integer suite, linux utilities, encryption and mediabench suites. These were divided into two equal sets of 12 benchmarks each. The first set is used as the *training set* (164.gzip, 181.mcf, 401.bzip2, 429.mcf, grep, lex, eqn, rc4, blowfish, cjpeg, pgpdecode, pgpen-code) to help in fixing the BERET design parameters and SEBs,

and the second set is used as the *test set* (175.vpr, 197.parser, 254.gap, 256.bzip2, 445.gobmk, cmp, yacc, sha, pc1, idct, gsmdecode, gs-mencode) to evaluate the benefits on an unknown set of application traces.

### 4.2 Hot Trace Coverage

The fraction of a program's execution time spent in the hot traces determines the overall benefits from utilizing the BERET hardware. In our experiments, we used a static compiler analysis implemented in the Trimaran compiler to identify hot traces. The results are shown in Figure 8 for all benchmarks in training set (first block of 12 bars), and test set (second block of 12 bars). Almost all the benchmarks were found to spend at least 15-20% of their execution time in hot traces, with many spending as much as 90%. The number of hot traces per benchmark varied from just a couple of them to as many as 50. Also note that while we use a static compilation flow in our evaluation, it is our belief that a better hot trace coverage can be garnered by a dynamic compilation framework. That would translate into even higher energy savings.

### 4.3 Determining SEB Collection

The objective of this study was to define the smallest collection of SEBs that exhibit a good mapping behavior for the traces in our benchmarks. Where, a good mapping implies that traces get divided into a small number of large subgraphs. Large subgraphs are better as they imply fewer CRAM accesses, more internal data forwarding, less number of register file accesses, and overall better energy savings. However, there is a trade-off here, because as an SEB gets larger, it also loses its flexibility across traces, forcing the need to have an overall bigger collection of SEBs.

We resolve this situation by performing a subgraph exploratory study on all traces in our training set of applications. The entire training set of applications is compiled, and subgraphs are enumerated from all recurring traces. From this list, we selected top eight *specialized SEBs* based on their frequency of occurrence across all traces, while maintaining a diversity in their sizes. The ISA in-
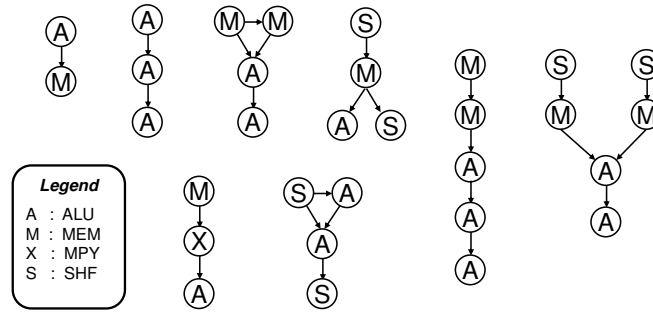
Figure 9: The final set of eight SEBs determined from the analysis of traces in training set. These are used for all evaluations of BERET design that follow in this paper.

structions were classified into four types (ALU, Shifter, Multiplier or Memory Access Unit) while forming these subgraphs. Figure 9 shows the final set of specialized SEBs.

For a given trace, the real metric to evaluate quality of specialized SEBs is the average subgraph size obtained for its mapping. Figure 10 shows these average subgraph sizes (across all traces) for every benchmark in training and test set. It is noteworthy that subgraphs sizes seen for applications in test set are at par with those seen for training set. This strongly suggests that specialized SEBs collection obtained using training set is stable for use in unknown general purpose applications.

## 4.4 Trace Characterization for Microarchitectural Parameters

Some of the important design parameters in the BERET hardware are the sizes of the CRAM, register file and store buffer. To determine a reasonable value for these parameters, we collected various statistics from the traces in our training set. For CRAM size, we analyzed the distribution of number of subgraphs per trace. As much as 90% of the traces had number of subgraphs less 12. Consequently, we fixed the CRAM size at $16 \times 64$ bits (our subgraph encoding fits in roughly 64 bits). For register file size, we analyzed the distribution of maximum live values per trace. This led to a register file with 8 entries. Finally, for the store buffer sizing, we looked at the distribution of store operations per trace, leading us to a store buffer with 6 entries.

## 5. EVALUATION

## 5.1 Methodology

In order to evaluate the potential of the BERET design, we used a comprehensive methodology involving compiler analysis for the identification and mapping of traces, an architectural simulator for performance, CAD tools for synthesis, place and route, power, area and finally, an energy simulator for computing total energy consumption while running a trace on BERET. Details about each of these components, along with benchmarks and baseline description follows below.

**Benchmarks:** A unique attribute of the BERET architecture is its relevance to both irregular as well as regular code based applications. The benchmark set was chosen to represent both these classes. We selected nine benchmarks from the SPEC integer suite (164.gzip, 175.vpr, 181.mcf, 197.parser, 254.gap, 256.bzip2, 401.bzip2, 429.mcf, 445.gobmk), five linux utilities (grep, cmp, lex, yacc, eqn), four encryption kernels (rc4, pc1, blowfish, sha) and six benchmarks from the MediaBench suite (cjpeg, idct, gsmdecode, gsmen-

code, pgpdecode, pgpencode). The division of these benchmarks into training and test set is discussed in Section 4.

**Baseline Processor:** The ARM1176 [1], a widely used processor in cellphones and portable electronics, was chosen to be the baseline processor for comparison. Being a single-issue in-order pipeline, we consider the ARM1176 to be an aggressive baseline for showing energy efficiency improvements. According to the ARM website [1], an 800MHz ARM1176 synthesized at 65nm technology node consumes roughly 160mW, which includes 16 KB level 1 instruction and data caches.

**Compiler Infrastructure:** The Trimaran compilation system [25] was used to implement the compiler flow that identifies hot traces and maps them to the BERET hardware. The trace identification component was implemented in OpenIMPACT (the front-end and profiling engine of Trimaran), whereas, the hardware mapping algorithms were implemented under ELCOR (back-end of Trimaran).

**Performance Simulation:** Cycle accurate simulators were used to model the performance of the baseline processor, as well as the execution time of traces mapped onto BERET. For the baseline single-issue in-order processor, we used SIMU, a performance simulator which is a part of the Trimaran package. A separate trace-based performance simulator was developed to measure the runtime of traces on the BERET hardware. This also accounted for the cost of execution control transfers between the main processor and BERET.

**Power and Area Estimation:** We implemented the BERET hardware in Verilog, and used a full CAD flow to synthesize (Synopsys Design Compiler), place and route (Cadence Encounter) and estimate power (Primetime PX). This was performed at the IBM 65nm technology node, while targeting a clock period of 1.25ns. This analysis gave us the power and area for all structures in the BERET hardware. Cache access power was estimated separately using CACTI [15] on a 16 KB, 4-way set associative cache.

**Energy Simulator:** The energy simulator was modeled after the BERET performance simulator. During the execution of a trace, it accumulates the energy consumed based on the number of activations for structures within BERET. For every activation, the average power for the structure is extracted from the CAD synthesis. Further, clock gating is assumed for the main processor whenever the control transfers to BERET. As a result, the processor logic incurs leakage energy in all phases of execution.
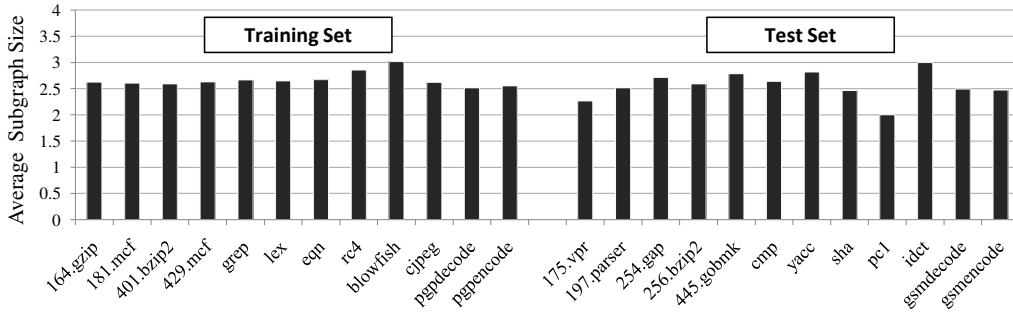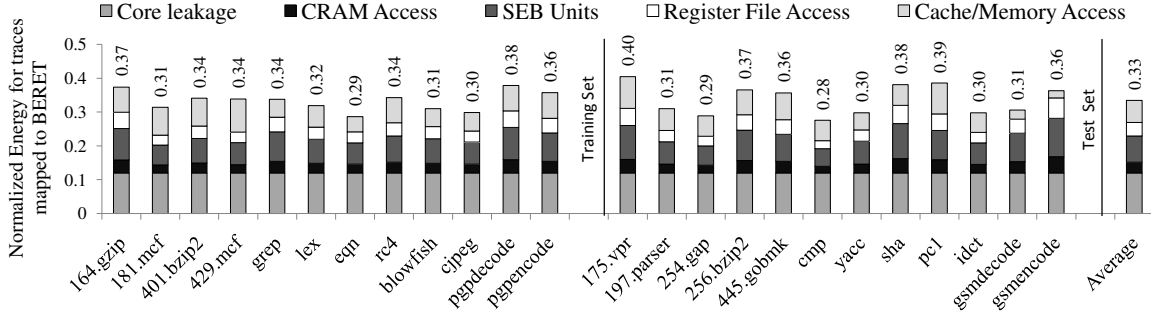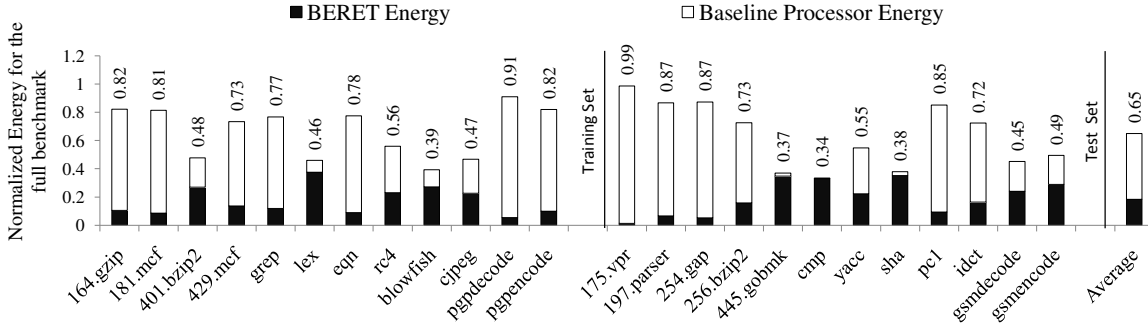
Figure 10: Average subgraphs sizes for training set and test set traces mapped to the specialized set of SEBs.



(a) Hot trace energy consumption while they ran on the BERET hardware (normalized to main processor).



(b) Full benchmark energy savings while using the BERET hardware in conjunction with the main processor.

Figure 11: Energy consumption relative to the baseline.

## 5.2  Results

### 5.2.1  Energy Savings

Figure 11a shows the normalized energy dissipation for the program regions running on BERET. The numbers shown also take into account the energy for transferring data and control in/out of the BERET hardware. The saving for a given application using BERET is tightly correlated with the average subgraph size seen for the same (see Figure 10). A larger subgraph size translates into higher savings, and vice versa. On average, the proposed design reduces energy by a factor of 3X over a single-issue in-order core. For reference, a carefully designed ASIC can give anywhere between 10-50X energy reduction for regular codes. However, unlike BERET, they are not programmable across applications. Further, for irregular codes, ASICs cannot be expected to reach the same level of efficiency due to their diverse control and memory access patterns. The absolute energy dissipation by the BERET hardware is roughly 65pJ per instruction.

The breakdown in the bars depicts the leakage energy of main core, and energy spent by various structures within BERET. The contribution of main core's leakage energy remains constant across all applications at about 12%. For BERET structures, the distribution of energy indicates a higher fraction spent for actual compute (SEB Units), relative to instruction and data supply. This is a notable improvement over the distribution shown in Figure 2, and it follows from the fact that energy savings in BERET are focused around the instruction supply (fetch, decode) and register data supply (fewer registers, reduced temporary variable accesses), with peripheral benefits from having smaller structures and eliminating pipeline latches. Just as in the case of average subgraph sizes, the results are comparable between training and test sets.

Figure 11b shows the energy numbers for the complete application runs. In this case, the overall benefits are correlated to the fraction of a program covered by hot traces (see Figure 8). The program portions that get mapped to the BERET hardware (black) garner significant energy savings, while the rest of them (white) dissipate
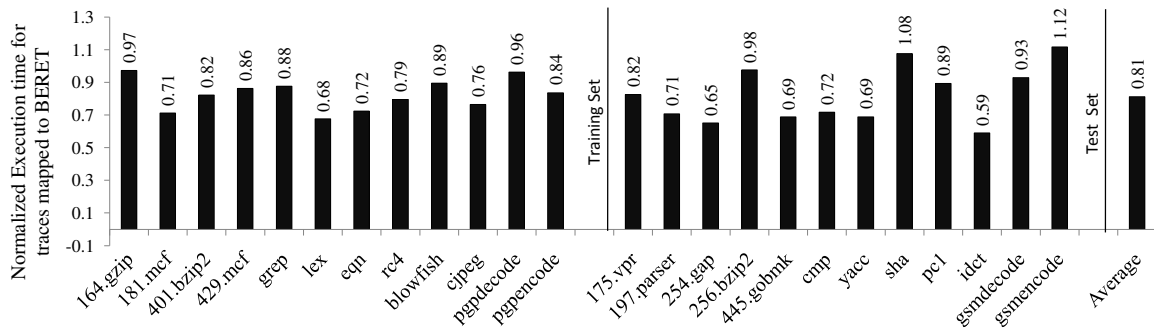
(a) Hot trace execution time while they ran on the BERET hardware (normalized to the main processor).
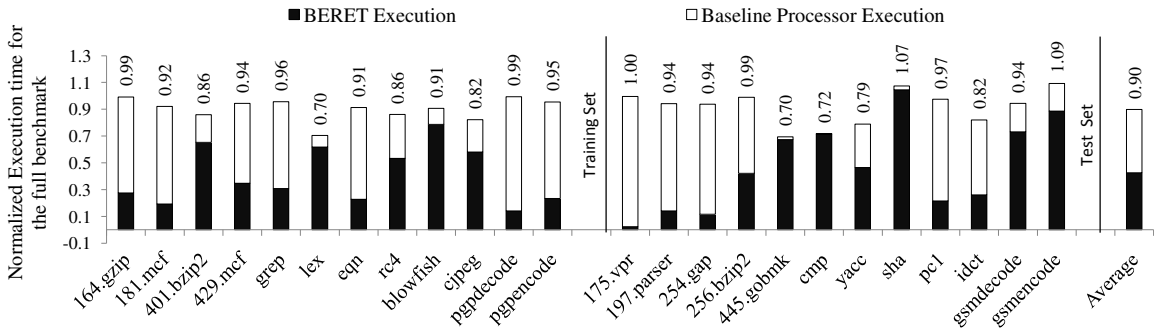


(b) Full benchmark execution time while using the BERET hardware in conjunction with the main processor.

Figure 12: Execution time relative to the baseline.

the standard energy on the main processor. The full benchmark energy savings ranged from 1% on 175.vpr to 66% on cmp, with an average of 35%.

### 5.2.2 Performance Comparison

The primary objective of the BERET design was to target energy savings in irregular codes, without sacrificing any performance. Fortunately, the use of a bulk execution model, using subgraphs instead of isolated instructions, gives a performance edge to BERET in certain cases. Figure 12a shows the normalized execution time for code regions mapped to the BERET hardware. On average, benchmark traces exhibit 19% performance improvement. This improvement stems from the instruction level parallelism (ILP) achieved within an SEB as it executes a data flow subgraph. For instance, a subgraph containing two `add` instructions feeding their outputs to a `xor` instruction would finish in two cycles, resulting in an IPC of *1.5*. For a minority of benchmarks, a performance loss is also seen. This results in cases where ILP benefits do not successfully outweigh the overheads of communication and early exits when using the BERET hardware.

Figure 12b shows the execution time improvement for the entire benchmark execution. As in the case of energy savings, the performance impact of using BERET gets diluted in accordance with the fraction of hot trace coverage. Overall, the execution time for the benchmarks evaluated is reduced by 10%.

### 5.2.3 Design Overheads

The final design of the BERET architecture consisted of a 128 byte CRAM, 8-entry register file, 6-entry store buffer, a heterogeneous set of 8 SEBs, and miscellaneous logic and interconnects. The total area for this in the 65nm technology node (after place and

route) was $0.396mm^2$. In the same technology node, the area of an ARM1176 core is $1.94mm^2$.

The BERET hardware is practically a standalone engine. As a result, it does not prolong any critical path in the base processor, and is expected to maintain the original operational frequency for the same.

## 6. RELATED WORK

The architectural designs for performance and energy have been an active area research for a long time. In this landscape (see Table 1), BERET stands out by being the only general purpose compute engine that provides high energy-efficiency for irregular (e.g., desktop workloads and SPEC int) as well as regular codes (e.g.,media kernels). Further, the BERET design can be attached as a co-processor to the main core, without any elaborate hardware or programming paradigm changes.

Specialized hardware designs [16, 18, 21] and instruction set extensions [9, 24] have long been a source of performance and energy efficiency for computations such as media kernels [13], encryption, signal processing [11]. ASIC designs are a good example of this, and get on the order of 40-50X energy efficiency improvements over simple RISC processors. Loop accelerator (LA) [7] designs are a limited form of ASICs that target modulo-schedulable, regular loop bodies with highly predictable memory access patterns. More recently, some flexibility has also been incorporated in these LAs [6, 30, 4] to generalize them for more than one application. BERET differs from such LAs and traditional ASICs in two ways: 1) it targets *irregular codes*, that are heavily control divergent, hard-to-parallelize, and not well-suited to modulo scheduling; 2) it is general purpose and not application specific.

Irregular codes have also been targeted by a recent work titled Conservation Cores (C-Cores) [**?**]. C-cores borrows insights from

Table 1: Comparison to Prior Work.

| | BERET | ASIC [16, 21] ASIP [9] | Loop Accelerators [7, 6, 29] | C-Cores [?] | ELM [5] | Programmable FUs [3, 19] |
|---|---|---|---|---|---|---|
| **Energy Savings** | High | V.High | V.High | High | V.High | Low |
| **Multiple Applications** | Yes | No | No | No | Yes | Yes |
| **Irregular Codes** | Yes | No | No | Yes | No | Yes |
| **Area** | Medium | Large | Medium | Large | Large | Small |
| **Processor Integration** | Co-processor | Stand-alone | Co-processor | Stand-alone | Stand-alone | In-pipeline |
| **Program Scope** | Traces | Full | Loops | Functions | Full | Op-chains |

prior spatial computation solutions [2] and synthesizes application-specific hardware for energy-efficiency improvements. However, this scheme requires an independent co-processor for every application, imposing heavy area and design time costs. In contrast, BERET engine is general purpose and not tied to any application or domain.

Trace Cache [12] and Loop Stream Detector (LSD) [22] are two popular industrial solutions introduced by Intel in Pentium 4 and Nehalem line of chips, respectively, for saving processor energy. Trace Cache is an SRAM structure that buffers sequences (called traces) of program instructions in a pre-decoded form, and in case of a hit, the back-end directly reads instruction from here. This removes the activation cost of predictors and decoders, while also garnering speed-up. BERET differs with Trace Cache on two fronts: 1) BERET eliminates instruction cache access, whereas due to its large SRAM structure, read access for Trace cache is similar to the original instruction cache, and 2) BERET uses a bundled execution model, and saves back-end data supply energy, whereas Trace cache provides no such savings for back-end energy.

The LSD design is conceptually very similar. However, instead of storing traces, it can buffer loops with fewer than 28 micro-ops (compare this to Trace Cache that can store 12-K micro-ops). For any program loop that can be accommodated within LSD, instruction fetch and decode energy is saved. Unfortunately, a large fraction of loops in irregular codes are large, and cannot fit inside LSD. Further, just like Trace Cache, LSD still incurs the energy expenses from inefficiencies in the processor back-end.

Another approach for irregular codes has been the use of subgraph accelerators like CCA [3], PRISC [19] to improve their performance. These solutions propose adding a customizable functional unit within the processor, that can improve performance for a range of data flow subgraphs encountered during a benchmark run. Unfortunately, the efficiency gains from these schemes are limited as they target primarily the back-end energy savings (data supply). The instruction supply still has to perform redundant fetches and decodes for the custom instruction(s). On the other hand, BERET targets both instruction and data supply savings.

DySER [?] is a recent proposal that builds upon the concept of programmable unit, and introduces DySER block, a grid of networked functional units. The approach works by splitting the original program into two streams: 1) compute instructions for DySER block, and 2) memory and control instructions for execution on the main pipeline. Although an interesting framework, the savings are limited due to frequent data and control communication between DySER block and main pipeline. Further, unlike BERET, it does not save on instruction supply energy.

ELM [5] is a programmable processor design dedicated to both instruction and data supply energy savings. While it achieves considerable efficiency improvements, the targeted applications are regular kernels from the embedded systems.

The BERET design bears some resemblance to data flow machines, as it breaks down the recurring traces from a benchmark into constituent data flow subgraphs. However, the full blown data flow designs such as WaveScalar [23] and TRIPS [20] are more performance centric, and introduce large area and complexity costs. Another related effort is the Braids [26] architecture, that converts the pipeline back-end into a series of homogeneous execution units, called braid execution units (BEUs). However, unlike BERET, the Braids architecture is performance centric, and works towards achieving aggressive issue-widths in simple in-order cores.

Finally, reconfigurable architectures have been used in the past for performance improvements. Garp [10] and Chimaera [28] use an FPGA-like substrate to map instruction sequences. Garp can also handle tight inner-most loops from an application. However, the use of a reconfigurable fabric, and dependence on regular code behavior limits their overall usability and impact on general purpose energy efficiency.

# 7. CONCLUSION

With the growing importance of energy conservation in all domains of computing, there is a clear need for architects to develop efficiency solutions that apply to general purpose computing. This is especially true given that the embedded systems approach of designing special purpose hardware does not scale to the requirements of irregular and diverse code base in general purpose application space. Towards this end, this paper identified the challenges posed by irregular codes, and developed BERET, an energy-efficient architecture for general purpose programs. Further, the BERET architecture is not application specific and can be programmed to deliver efficiency improvements for virtually any recurring trace of instructions. Fundamentally, BERET relies on these recurring traces to cut down on redundant instruction fetch and decode energy, and a bundled execution model to reduce register file access energy. The capability to handle multiple applications and offload long ranges of instructions, sets BERET apart in the space of efficiency solutions.

We applied BERET on a variety of benchmarks from SPEC integer suite, Linux utilities, and MediaBench. On average, we found that BERET can reduce energy by a factor of 3X for the program regions it executes. The average energy savings for the entire application was 35% over a single-issue in-order processor.

# 8. ACKNOWLEDGMENTS

# 9. REFERENCES

[1] ARM. Arm11.
http://www.arm.com/products/CPUs/families/ARM11Family.html.

[2] M. Budiu, G. Venkataramani, T. Chelcea, and S. C. Goldstein. Spatial computation. In *12th International*

*Conference on Architectural Support for Programming Languages and Operating Systems*, pages 14–26, 2004.

[3] N. Clark et al. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proc. of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Dec. 2004.

[4] N. Clark, A. Hormati, and S. Mahlke. VEAL: Virtualized execution accelerator for loops. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 389–400, June 2008.

[5] W. J. Dally, J. Balfour, D. Black-Shaffer, J. Chen, R. Harting, V. Parikh, J. Park, and D. Sheffield. Efficient embedded computing. *IEEE Computer*, 41(7):27–32, July 2008.

[6] K. Fan, M. Kudlur, G. Dasika, and S. Mahlke. Bridging the computation gap between programmable processors and hardwired accelerators. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 313–322, Feb. 2009.

[7] K. Fan, M. Kudlur, H. Park, and S. Mahlke. Compiler-directed synthesis of multifunction loop accelerators. In *Proc. of the 2005 Workshop on Application Specific Processors*, pages 91–98, Sept. 2005.

[8] D. Friendly, S. Patel, and Y. Patt. Putting the fill unit to work: Dynamic optimizations for trace cache microprocessors. In *Proc. of the 25th Annual International Symposium on Computer Architecture*, pages 173–181, June 1998.

[9] R. Hameed, W. Qadeer, M. Wachs, O. Azizi, A. Solomatnikov, B. C. Lee, S. Richardson, C. Kozyrakis, and M. Horowitz. Understanding sources of inefficiency in general-purpose chips. In *Proc. of the 37th Annual International Symposium on Computer Architecture*, pages 37–47, 2010.

[10] J. R. Hauser and J. Wawrzynek. GARP: A MIPS processor with a reconfigurable coprocessor. In *Proc. of the 5th IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 12–21, Apr. 1997.

[11] T. Instruments. Tms320c2x user's guide, Jan. 1993.

[12] Intel. Intel xeon processor with 512 kb l2 cache, 2004.

[13] H. Kalva. The H.264 video coding standard. *IEEE MultiMedia*, 13(4):86–90, 2006.

[14] W. mei W. Hwu, S. A. Mahlke, W. Y. Chen, P. P. Chang, N. J. Warter, R. A. Bringmann, R. G. Ouellette, R. E. Hank, T. Kiyohara, G. E. Haab, J. G. Holm, and D. M. Lavery. The superblock: An effective technique for vliw and superscalar compilation. *Journal of Supercomputing*, 7(1):229–248, May 1993.

[15] N. Muralimanohar, R. Balasubramonian, and N. P. Jouppi. Optimizing nuca organizations and wiring alternatives for large caches with cacti 6.0. In *IEEE Micro*, pages 3–14, 2007.

[16] M. Papadonikolakis et al. Efficient high-performance ASIC implementation of JPEG-LS encoder. In *Proc. of the 2007 Design, Automation and Test in Europe*, pages 159–164, Apr. 2007.

[17] S. J. Patel and S. S. Lumetta. rePLay: A hardware framework for dynamic optimization. *IEEE Transactions on Computers*, 50(6):590–608, June 2001.

[18] P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavorial synthesis of ASICs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 8(6):661–679, June 1989.

[19] R. Razdan and M. D. Smith. A high-performance microarchitecture with hardware-programmable function units. In *Proc. of the 27th Annual International Symposium on Microarchitecture*, pages 172–180, Dec. 1994.

[20] K. Sankaralingam et al. Exploiting ILP, TLP, and DLP using polymorphism in the TRIPS architecture. In *Proc. of the 30th Annual International Symposium on Computer Architecture*, pages 422–433, June 2003.

[21] R. Schreiber et al. PICO-NPA: High-level synthesis of nonprogrammable hardware accelerators. *Journal of VLSI Signal Processing*, 31(2):127–142, 2002.

[22] R. Singhal. Inside intel next generation nehalem microarchitecture, 2008. http://software.intel.com/file/18976.

[23] S. Swanson, K. Michelson, A. Schwerin, and M. Oskin. Wavescalar. In *Proc. of the 36th Annual International Symposium on Microarchitecture*, page 291. IEEE Computer Society, 2003.

[24] Tensilica Inc. *Diamond Standard Processor Core Family Architecture*, July 2007. http://www.tensilica.com/pdf/Diamond WP.pdf.

[25] Trimaran. An infrastructure for research in ILP, 2000. http://www.trimaran.org/.

[26] F. Tseng and Y. N. Patt. Achieving out-of-order performance with almost in-order complexity. In *Proc. of the 35th Annual International Symposium on Computer Architecture*, pages 3–12, June 2008.

[27] M. Woh et al. From SODA to scotch: The evolution of a wireless baseband processor. In *Proc. of the 41st Annual International Symposium on Microarchitecture*, pages 152–163, Nov. 2008.

[28] Z. A. Ye et al. CHIMAERA: a high-performance architecture with a tightly-coupled reconfigurable functional unit. In *Proc. of the 27th Annual International Symposium on Computer Architecture*, pages 225–235, 2000.

[29] S. Yehia et al. Exploring the design space of LUT-based transparent accelerators. In *Proc. of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, pages 11–21, Sept. 2005.

[30] S. Yehia, S. Girbal, H. Berry, and O. Temam. Reconciling specialization and flexibility through compound circuits. In *Proc. of the 15th International Symposium on High-Performance Computer Architecture*, pages 277–288, 2009.