The Precomputed Branch Architecture

Brad Calder

Dept. of Computer Science and Engineering
University of California, San Diego
9500 Gilman Drive
La Jolla, CA 92093-0114
calder@cs.ucsd.edu

Dirk Grunwald

Dept. of Computer Science
University of Colorado
Campus Box 430
Boulder, CO 80309-0430
grunwald@cs.colorado.edu

UCSD Technical Report
CS97-526, March 1997

Abstract

Accurate instruction fetch and branch prediction is increasingly important on today's superscalar architectures. Fetch prediction is the process of determining the next instruction to request from the memory subsystem. Branch prediction is the process of predicting the likely out-come of branch instructions. A branch target buffer (BTB) is often used to provide target addresses for taken branches and to predict the destination of indirect jumps. Using a BTB avoids the delay needed to recalculate the destination address and reduces the misfetch penalty. However, an effective branch target buffer can be large and can possibly increase the cycle time of a processor.

We propose that a design used in older computers, such as the PDP-8, be used in modern architectures instead of a BTB design. The compiler would pre-compute the branch destination for most branch instructions, allowing the branch information to be stored with the instruction. We consider computing branch destinations at link time and as instructions are fetched into the instruction cache; both alternatives offer similar performance with different advantages. A very small branch target buffer is still useful to predict indirect branches, which can not be pre-computed. Our results show that the *Precomputed-Branch* architecture performs better than an architecture using only a branch target buffer, and has significant hardware savings. This is particularly true for larger programs more representative of modern applications.

1 Introduction

Modern superscalar processor designs are extremely sensitive to control flow changes. Changes in control flow, be they conditional or unconditional branches, direct or indirect function calls, or returns can significantly hinder the performance of a processor. To keep the pipeline fully utilized, past processors typically fetched the address following the most recent address, because the target address for a program counter relative (PC-relative) branch is typically not available until the instruction is decoded. If the decoded instruction is a break in control flow, the previously fetched (fall-through) instruction can not be used, and

a new instruction must be fetched after the target address is calculated, introducing a pipeline bubble or unused issue slots.

The final destination for conditional branches, indirect function calls and returns are typically not available until a later stage of the pipeline. The processor may elect to fetch and decode instructions on the assumption that the eventual branch target can be accurately predicted. If the processor mispredicts the branch destination, instructions fetched from the incorrect instruction stream must be discarded, leading to several wasted issue slots. This is called a branch *mispredict penalty*. To eliminate this mispredict penalty current processors use *branch prediction* architectures. Branch prediction is the process of predicting the direction for conditional branches (taken or not-taken) and the destination for indirect branches and return instructions.

Even if a conditional branch is correctly predicted as taken, the target instructions cannot be correctly fetched from the instruction cache in the next cycle unless the processor knows the target address of the branch. Since the target address is not calculated until the decode stage, this causes a pipeline stall called a *misfetch penalty*. The misfetch penalty is caused by not knowing the taken target address for a PC-relative branch the same cycle the predicted taken branch instruction is fetched from the cache. To eliminate the misfetch penalty, *fetch prediction* is used along with branch prediction to predict which instructions to fetch each cycle from the instruction cache. Fetch prediction architectures, such as a branch target buffer, provide mechanisms to predict the taken target addresses for branch instructions effectively eliminating the misftech penalty.

In practice, pipeline bubbles due to mispredicted breaks in control flow degrade a programs performance more than the misfetch penalty. Though, as processors issue more instructions concurrently, these penalties increase, and the instruction misfetch penalty becomes increasingly important. It is more likely that a branch will occur as more instructions are fetched each cycle, decreasing the likelihood that the fall-through instruction will be executed. A branch target buffer (BTB) is one mechanism for effectively eliminating misfetch penalties by providing taken branch target addresses. When an instruction is fetched, the same address is offered to the BTB; if there is a match in the BTB, the next instruction is fetched using the target address specified in the BTB if the branch is predicted as taken. In this design, exemplified by the PowerPC 604 [33], the BTB identifies the instruction as a branch and only records the destination for taken branches.

1.1 The Branch Target Address Problem and the Precomputed-Branch Solution

Architectures using BTB's can issue a large number of instructions per cycle because of accurate branch and fetch prediction. However, BTB's can lead to a complex architecture and large BTB's can be costly to implement. In this paper, we show how to achieve the same or better performance using simpler techniques.

We show that we can maintain a low branch execution cost with considerably fewer resources than that needed by architectures using a branch target buffer. Our proposed architecture assumes a flat address space, and eliminates program-counter relative (PC-relative) branches. In this design, the destination address is quickly computed by concatenating the branch's displacement, which is pre-computed at compile time, with the high-order bits of the current address, effectively dividing memory into a number of program segments or *branch spaces*. Branches between branch spaces are computed and performed as indirect jumps. This design assumes the program linker or compiler can partition the program into a number of segments and modify the program's structure to select between intra-space and inter-space branches [35]. We also describe how existing architectures using PC-relative branches can be extended to benefit from pre-computed branches.

The Precomputed-Branch architecture provides area-efficient support for conditional, unconditional and procedural branches, where the branch destination is explicitly specified. Other researchers have described inexpensive mechanisms to predict the destination of procedure returns [19]. The only remaining branch type requiring a predicted target address for the Precomputed-Branch architecture is an indirect jump, where the destination may be specified by values computed during execution. We show that using a very small branch target buffer, dedicated to predicting only the outcome of indirect jumps, benefits a number of programs for the Precomputed-Branch architecture. This is especially true for object-oriented programs containing a large number of indirect jumps.

We originally proposed the use of a Precomputed-Branch architecture in [5]. In this previous study we focused on improving the performance of the BTB design by examining different BTB configurations and update policies, and in addition we proposed the Precomputed-Branch architecture as an alternative to the BTB. In that study we examined the performance of only 11 programs and the use of only static prediction to predict indirect jumps when using the Precomputed-Branch architecture. We did not examine the compiler support needed for the Precomputed-Branch architecture, nor did we examine dynamic mechanisms for predicting the indirect jump created by the Precomputed-Branch architecture. We also did not examine the hardware costs and cycle time implications of the Precomputed-Branch architecture, nor did we discuss how to apply this technique to current PC-relative instruction set architectures.

In this paper we expanded our study to 35 programs, and we study the performance of using a small sized branch target buffer to predict the direction for indirect jumps when using the Precomputed-Branch architecture. In addition we expand on our previous study by examining several algorithms for compiler partitioning of a program into branch spaces, and we measure the effect of these algorithms on a number of architectural configurations and different branch segment sizes. Partitioning programs into branch spaces introduces new indirect jumps to span across branch spaces. We measure the number of such branches introduced, and other overhead introduced by the partitions. We also examine the hardware costs of the Precomputed-Branch architecture in comparison to the BTB architecture. Lastly, we discuss how existing superscalar architectures can be modified to use the Precomputed-Branch architecture.

In the next section, we examine previous work in branch prediction and branch target buffer design. In §3, we describe a branch target buffer architecture, used in our experiments to compare against the performance of the Precomputed-Branch architecture, also described in §3. Section 4 describes the structure of the execution-driven simulation study and the performance metrics we used. The Precomputed-Branch architecture requires programs to be partitioned into branch spaces, and §5 describes algorithms to accomplish this and their performance. Section 6 compares the performance of the BTB and Precomputed-Branch architectures, and some practical issues are discussed in §7. We conclude in §8.

2 Background

There are two sources of pipeline stalls we want to remove, branch misfetch and mispredict penalties. A branch target buffer can be used to reduce the misfetch penalty and can be used as a simple branch prediction mechanism. Other branch prediction methods can reduce mispredict penalties, but not misfetch penalties. Many architectures [33, 40, 42] combine branch target buffers and other branch prediction mechanisms to reduce both misfetch and mispredict penalties.

2.1 Branch Target Buffers

Misfetch penalties can be reduced in a number of ways, such as using branch delay slots [24], a table of cache indices for fetch prediction [8], or branch target buffers [21, 22, 29, 32]. A BTB can eliminate misfetch stalls by storing the branch destination. For unconditional branches, indirect jumps or functions calls, this destination can be immediately fetched. For conditional branches, either the fall-through or the destination stored in the BTB is fetched. Some form of branch prediction is needed to select between the fall-through and taken address for conditional branches.

Typically, a BTB contains from 32 to 512 entries with varying degrees of associativity. A BTB requires considerable storage, because it stores the address of the branch as the tag *and* the address of the probable destination. Different kinds of branches use different mechanisms to predict their branch destinations. To be able to select among the different mechanisms for the different branches, we need to be able to identify the branch type, and some BTB designs store the branch type in the BTB. For function calls (either direct or indirect), the previous function address is stored in the 'destination' field of the BTB. This can also be done for return instructions, but a return stack is much more accurate [19]. In this study we assume the branch type is stored in the instruction cache or is easily identifiable in the branch's instruction encoding, so it is not stored in the BTB.

2.2 Branch Prediction Mechanisms

The other component of most branch architectures is some mechanism to predict whether conditional branches are taken or not-taken. Branch prediction techniques are classified as *static* or *dynamic*. Static branch prediction information does not change during the execution of a program, while dynamic prediction may change, reflecting the time-varying activity of the program. Static methods range from compile-time heuristics [3, 9, 21, 24, 32] to profile-based methods [10, 14, 24, 37, 43]. In general, profile based prediction techniques outperform compile-time prediction techniques or techniques that use heuristics based on the direction of the branch target (forward or backward) or instruction opcode. While static prediction mechanisms, particularly profile-based methods, accurately predict 70-80% of branches, modern computer architectures increasingly depend on mechanisms that estimate future control flow decisions to increase performance, requiring more accurate branch prediction mechanisms.

A *pattern history table* (PHT) is a mechanism for predicting conditional branches. It does not store the site (tag) and target addresses of branches as in the BTB; rather, the table only stores N-bit counters used to predict the direction for conditional branches. The most common variants of this design are 1-bit counter techniques that indicate the direction of the most recent branch mapping to a given prediction bit, and 2-bit counter techniques that yield much better performance for programs with loops [21, 24, 32]. These designs use the branch site address as an index into the PHT. Since different branch addresses can index into the same table entry, several conditional branches may share the same prediction information. For example, in a 4096 entry table, branches at addresses 0, 16384 and 32768 all map to the same entry in the table. When a conditional branch at these addresses is executed, the information for entry '0' is used to predict the branch direction, even if that information was recorded for one of the other branches. The advantage of the pattern history tables is that they keep track of very little information per conditional branch site and are very effective in practice.

More recently Pan *et al* [28] and Yeh and Patt [40, 42] have proposed *branch-correlation* or *two-level* branch prediction mechanisms. Although there are a number of variants, these mechanisms generally

combine the history of several recent branches to predict the outcome of an incipient branch. The simplest example is the so-called *degenerate method* of Pan *et al.* When using a 4096 entry table, the processor maintains a 12-bit shift register (the global history register) that records the outcome of previous branches. If the previous 12 branches that executed were a sequence of three taken branches, six non-taken branches and three more taken branches (TTTNNNNNNTTT), the register might store the value 111000000111_2, or 3591. This is used as an index into the 4096-entry table, much as the program counter is used in the prediction history table method. This provides contextual information about particular patterns of branches. Some methods combine the history register with other information. For example, McFarling [23] used an exclusive-or of the program counter and the global history register to scatter the table references, improving the PHT's performance.

2.3 Combining Branch Target Buffers and Branch Prediction

Originally, BTB's were used as a mechanism for branch prediction, effectively predicting the prior outcome of a branch [21, 29, 32] and providing the target address. Researchers have proposed associating additional branch prediction information with each BTB entry to improve branch prediction [41], and a variation of this technique has been implemented in the Intel Pentium and PentiumPro architectures. The problem with this technique is the branch prediction information can only be used on a BTB hit, or when when a branch address is found in the BTB. We call designs that associate branch prediction information with the branch target buffer a *coupled* branch architecture, since the conditional branch prediction information is associated with the BTB and can only be used if a branch hits in the BTB. In a related paper [5], we showed that *decoupled* branch architectures can provide slightly better performance than coupled branch architectures. A decoupled architecture separates the conditional branch prediction information from the branch target buffer, so that it can be used to correctly predict a branch even when that branch is not in the BTB. In the remainder of this paper, we only consider decoupled branch architectures.

3 The Design of Two Branch Architectures

We used trace-driven simulation to compare the Precomputed-Branch architecture to a design that makes aggressive use of branch target buffers. We simulated the decoupled branch architecture proposed in [5], because this architecture provides better overall branch performance than the coupled models proposed in [41] for the design space considered in this paper. In this section, we describe this architecture and follow that with a detailed description of the Precomputed-Branch architecture.

3.1 A BTB-based Instruction Fetch Architecture

Figure 1 is a schematic representation of a conventional branch prediction and instruction fetch architecture using a branch target buffer. A BTB is used to eliminate misfetch penalties by providing taken target addresses, while a pattern history table (PHT) is used to predict conditional branches. In this design, the global history register is combined with the program counter using an exclusive-or, and the result is used as the index into the PHT [23]. In both the BTB and Precomputed-Branch architecture, we assume the branch type is encoded in the instruction, or pre-decoded and stored in the instruction cache. A 32-entry return address stack is used to predict the outcome of return instructions.

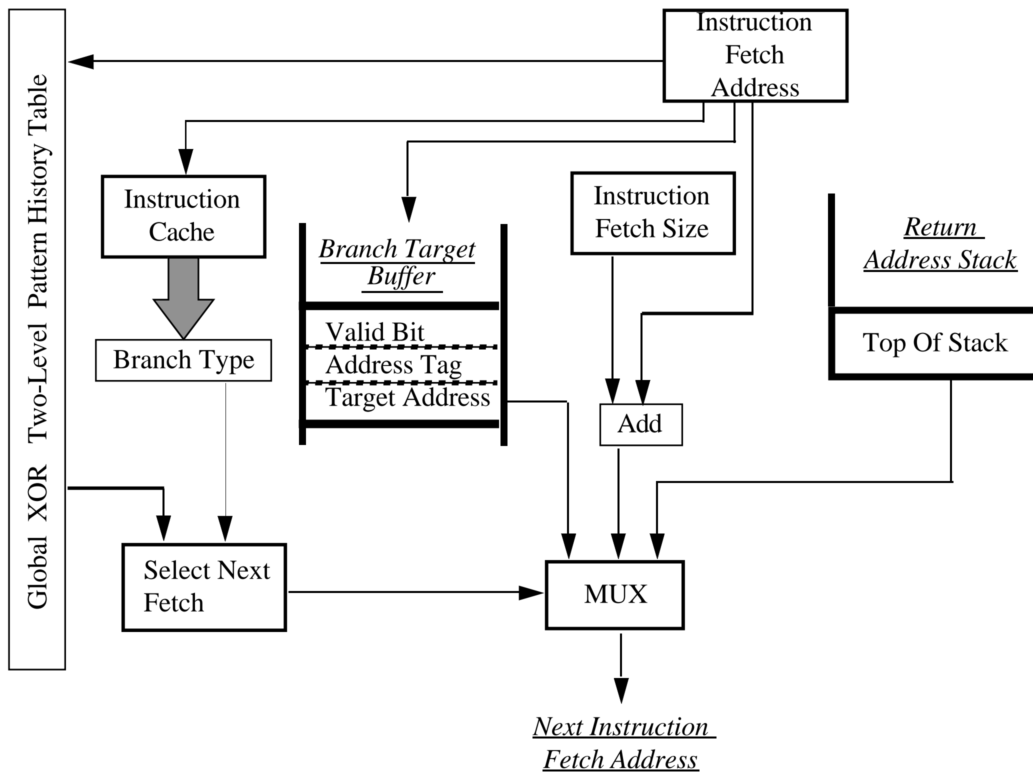


Figure 1: A Schematic Representation of a Branch Prediction Architecture Using a Decoupled Two-Level Pattern History Table and a Branch Target Buffer.

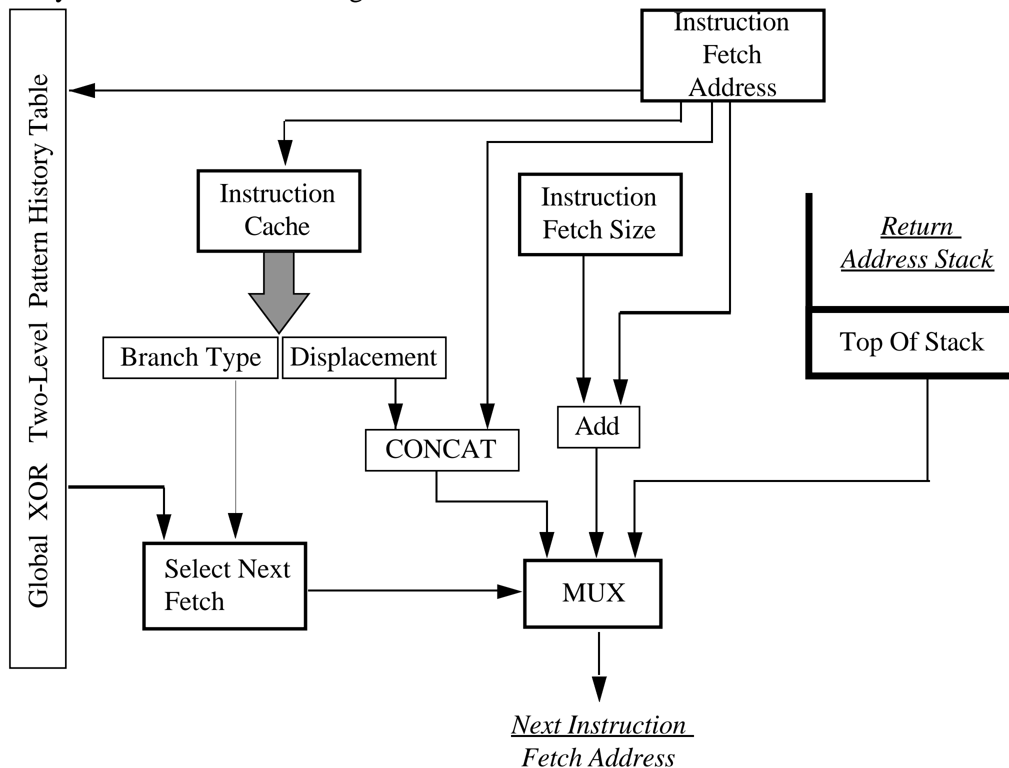


Figure 2: A Schematic Representation of the Precomputed-Branch Architecture Using a Two-Level Pattern History Table.

As shown in Figure 1, the current instruction address is concurrently offered to the instruction cache, providing the actual instruction, to the PHT, and to the BTB. There are three important types of branches: direct or indirect branches, conditional branches and returns. Depending on the branch type and the branch prediction information from the 2-level PHT, either the BTB target address, the computed fall-through address, or the return stack address is selected as the next instruction fetch. If a PC-relative branch misses in the BTB, the fall-through address is fetched. If this branch was predicted as taken by the decoupled PHT, then the taken address is fetched after it is calculated in the decode stage. Therefore, on a BTB miss, if the PHT correctly predicts the PC-relative branch as taken, only a misfetch penalty results.

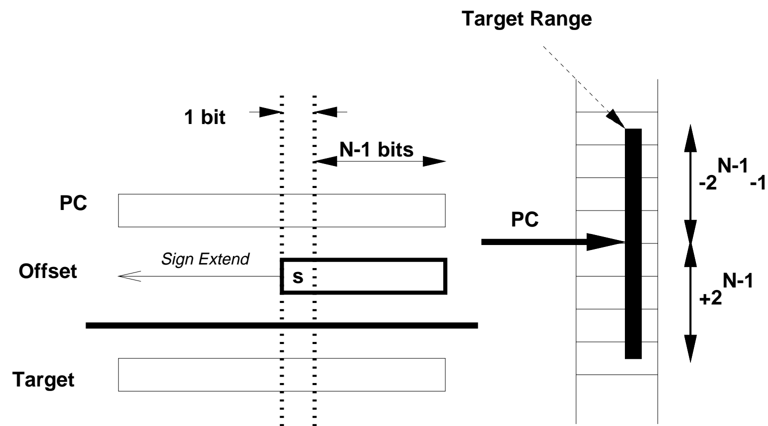
In this BTB architecture, when an unconditional branch is executed and there is a BTB miss, the BTB entry is updated to record the computed target address. When the branch is encountered again, and there is a BTB hit, the branch type indicates this is an unconditional branch, and the architecture uses the target address stored in the BTB for the next cache fetch. Conditional branches have similar actions; however, the prediction information from the PHT is used to predict the likely outcome of conditional branches. Depending on the predicted outcome, the stored destination (which is always the 'taken' address) in the BTB or the fall-through address is used to fetch the next instruction. When a return instruction is encountered the branch type indicates that the instruction is a return and the top of the return stack is used to fetch the next instruction. When an indirect jump is executed, if there is a BTB hit, the address stored in the BTB is used for the next cycle's instruction fetch.

3.2 The Precomputed-Branch Architecture

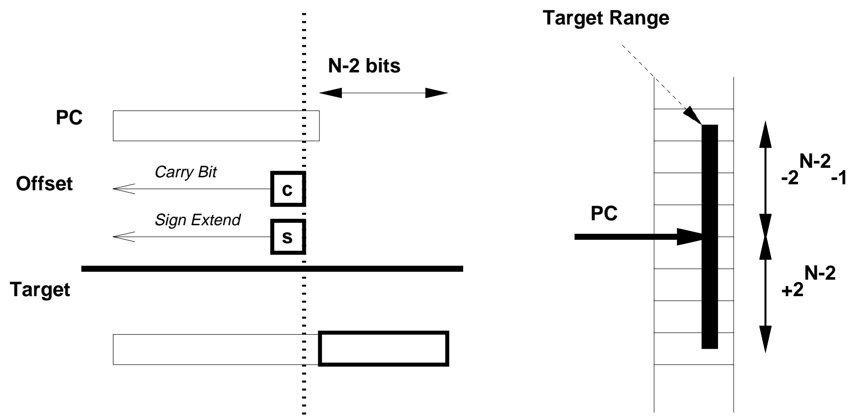
The BTB serves two roles. Only taken branches are entered in the BTB, so a BTB hit indicates the instruction is a branch, and the BTB provides the target address. As mentioned, we assume that the instruction cache or the instruction code can identify the branch type. This can either be made explicit in the instruction encoding, or the instruction can be partially decoded when it is brought into the instruction cache; a similar mechanism has been used in several architectures, such as the MIPS R10000 [26]. The only remaining function provided by the BTB is the pre-computed destination address for taken branches. The BTB is needed because the destination address specified by a branch instruction can not be fetched from the instruction cache and computed all in a single cycle when using PC-relative destinations.

Figure 2 shows our proposed instruction fetch architecture. A program is broken into multiple *branch spaces*. Branches within a single branch space can use a normal branch instruction, while branches between branch spaces must be computed as an indirect jump. The pre-computed displacement indicates the branch target displacement within the current branch space. The lower-order bits of the branch destination are simply concatenated with the higher-order bits of the current address, and no addition is needed. Issues surrounding branch spaces, and the complications that arise, are discussed later.

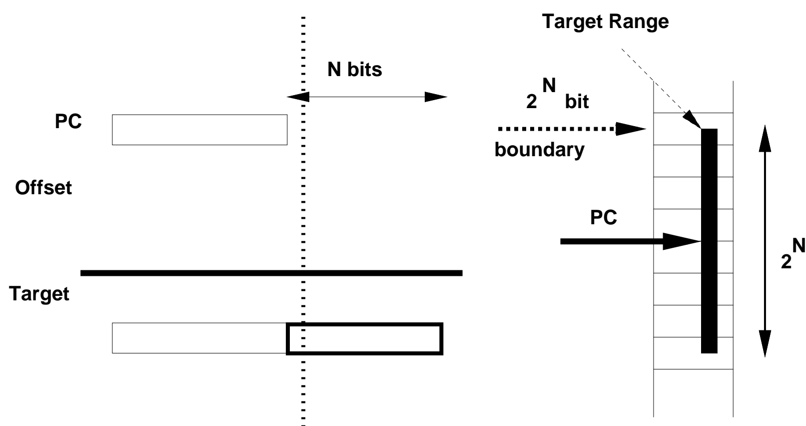
The concatenating of the pre-computed branch eliminates misfetch penalties for conditional branches, unconditional branches and direct procedure calls. This leaves indirect branches as the only branch type without a pre-computed branch address. Thus, we use a small BTB to predict indirect branches; otherwise they will always be mispredicted. Since this branch target buffer is only used for indirect jumps, we call this an *indirect jump buffer* (IJB) to clarify the distinction from the BTB architecture. Alternatively, we can use profile-based prediction of indirect function calls, which has been shown to be effective and important for the C++ programming language [7], where such branches occur frequently.



(a) Traditional Sign-Extended PC-Relative Branch



(b) Compiler-Assisted Sign-Extended Branch



(c) Precomputed-Branch (Not Sign-Extended)

Figure 3: Alternate Branch Methods

3.3 Computing the Branch Target

Traditional branch architectures use a PC-relative displacement; Figure 3(a), modeled after the diagrams in [20], schematically illustrates the process. In the encodings, information in lightly outlined boxes is provided or computed at execution time; for example, in Figure 3(a), the PC is available during execution. Heavily-outlined boxes show the information provided by the branch instruction – the instruction provides n bits for the branch displacement. On the right-hand side, the solid boxes show the range of instructions that can be addressed. For Figure 3(a), a displacement stored in the branch instruction is sign-extended to the size of the program counter and added to the program counter. Each branch can directly address instructions at address $PC - 2^{n-1} - 1 \ldots PC + 2^{n-1}$. For simplicity, we assume the program counter is always aligned on instruction boundaries, since we are chiefly concerned with architectures with fixed-width instructions.

Katevenis [20] proposed several branch encodings where the branch displacement field contains the least significant bits of the branch target address. Figure 3(b), shows one such encoding. Here, the sign bit for the offset and the carry for the addition of the lower bits are computed by the compiler (or linker) and encoded in the instruction. The lower bits can be immediately used to index the cache. Concurrent with the cache fetch, the higher order bits are computed and matched against the address tags when the cache fetch returns. If the tags are mismatched with the actual PC, an instruction-cache miss occurs and the pipeline is stalled. During the stall, the program counter is corrected. Since the instruction must include both the carry and the sign bit, an n-bit displacement can only address $PC - 2^{n-2} - 1 \ldots PC + 2^{n-2}$.

Figure 3(c) diagrams our proposed pre-computed branch encoding. We use an *explicit displacement* instead of a PC-relative displacement because we need to calculate target addresses in time to use them for the next instruction fetch and an adder is usually too complex for this purpose. The n-bit displacement is used as the lower order part of the destination address, and is concatenated with the higher order bits of the current PC to form the new fetch address. Each branch can then jump within a branch space of 2^n instructions. This breaks a program up into branch spaces or segments of size 2^n instructions. Every direct branch within a 2^n branch space can only branch within that space. To branch outside that span, an indirect jump must be used.

3.4 Other Non-Relative Branch Architectures

Using non-relative branches is not new idea, although we are unaware of studies with our emphasis on the branch layout algorithms, branch prediction architectures, and our analysis of modern programs.

The memory for the PDP-8 was divided into eight "memory fields," each field was divided into 32 pages of 64 locations (words). A `JMP` or `JMS` instruction could jump within a single memory page (e.g., to one of 64 words), or could specify an indirect reference to a word containing a 12-bit destination. Branches within the same page were pre-computed, and all other branches required an indirect reference. In the Crisp processor [12], a branch destination is included in every decoded instruction in the instruction cache, resulting in very large instructions – 192 bits. This technique consumes a considerable amount of space and may limit the processor cycle time. A mechanism similar to that used in the PDP-8 was used in Control Data and IBM processors, where instructions were optimized to execute within an instruction buffer – Lee and Smith [21] provide a good survey.

By comparison, we rely on the program linker to compensate for the limited branching by pre-computing branch destinations and reducing the number of complex operations (indirect branches). After a fashion,

we are applying the "RISC design philosophy" to branch architectures - we let the software (compiler and linker) share the burden of making the hardware efficient and inexpensive.

4 Experimental Methodology

We will pose several questions concerning branch architectures and answer those questions using information from trace-based simulation. We instrumented the programs from the SPEC92 benchmark suite, the Perfect-Club [4], and object-oriented programs written in C++. We used ATOM [34] to instrument the programs; due to the structure of ATOM, we did not need to record traces and could trace the complete execution of all the programs. The programs were compiled on a DEC 3000-400 using either the DEC FORTRAN, C, or C++ compiler. All programs were compiled with standard optimization (`-O`). We constructed several simulators to analyze the programs. The simulator was run once to collect information on call and branch targets, and a second time if we needed to use profile information from the prior run. For the SPEC92 programs, we used the largest input distributed with the SPEC92 suite.

The alternate programs include: `cfront`, version 3.0.1 of the AT&T C++ language preprocessor written in C++, `groff`, a version of the `ditroff` text formatter written in C++, `idl`, a C++ parser for the CORBA interface description language, `db++`, a version of the 'deltablue' constraint solution system written in C++, `lic`, a linear inequality calculator, and `porky`, a system performing many compiler optimizations. We selected these programs because we found that the SPECint92 suite did not typify the behavior seen in C++ programs [11], and our original goal was to understand the impact of branch architectures on C++ programs. For these alternate programs, we used sizable inputs that exercised a large part of the program.

Table 1 shows the basic statistics for the programs we instrumented. The table is divided into two sections; the first half shows dynamic information (the information gathered during a particular execution of the program), and the second part shows the static information. The static information is a property of the program binary, and is the same for all executions. The first three columns show the number of instructions traced, the percentage of breaks (e.g., conditional branches, return instructions and so on) encountered during execution and the percentage of conditional branches that are taken. The next five columns break down the number of breaks in control flow encountered during tracing into five classes: conditional branches (**CBr**), indirect jumps (**IJ**), unconditional branches (**Br**), procedure calls (**Call**) and procedure returns (**Ret**). The static information includes the total number of instructions and procedures found in each program.

Indirect jumps are used both to implement indirect function calls and some `switch` statements. Note that the C++ programs execute fewer conditional branches than C programs. In part, this is caused by the increased number of procedure calls, indirect jumps, and returns in the C++ programs. Also note that the C++ programs include many more procedures than the C and FORTRAN programs.

4.1 Performance Metrics

Our goal is to understand the performance improvement of various branch architectures; this requires a metric to compare one architecture to another. There are two forms of pipeline penalties: misfetch and misprediction penalties. Each branch type can be misfetched, but only conditional branches, indirect function calls and returns can be mispredicted. The penalty for misfetching is less than the penalty for misprediction. We may be willing to misfetch more branches if it means we can reduce the number of mispredicted branches. We record the percentage of misfetched branches (%MfB) and the percentage of

Program	Dynamic								Static	
	# Insn's Traced	%	% Taken	Percentage of Breaks During Tracing						
	(Millions)	Breaks	Cond. Br	%CBr	%Br	%IJ	%Call	%Ret	# Insn's	# Procs
APS	1,490	4.70	50.64	84.90	5.71	0.12	4.63	4.63	128,694	797
CSS	379	9.43	55.63	77.58	9.86	2.09	5.24	5.24	141,751	818
LGS	956	9.46	66.84	76.78	2.95	0.00	10.14	10.14	90,385	726
LWS	14,183	9.89	66.34	80.08	5.84	0.00	7.03	7.03	88,661	714
NAS	3,604	5.39	60.66	63.63	12.70	1.70	10.99	10.99	103,401	740
OCS	5,187	3.06	88.57	98.84	0.26	0.02	0.44	0.44	90,122	717
SDS	1,109	6.83	53.05	99.15	0.07	0.03	0.38	0.38	94,615	768
TFS	1,694	3.44	77.42	92.28	2.63	0.23	2.43	2.43	94,383	715
TIS	1,722	5.27	51.08	100.00	0.00	0.00	0.00	0.00	74,681	681
WSS	5,422	5.48	62.36	86.79	5.75	3.28	2.09	2.09	106,227	757
doduc	1,150	8.53	48.68	81.31	4.97	0.01	6.86	6.86	94,402	708
fpppp	4,333	2.82	47.74	86.66	8.01	0.00	2.66	2.66	83,999	685
hydro2d	5,683	6.28	73.34	95.84	1.38	0.00	1.39	1.39	85,808	716
mdljsp2	3,344	10.60	83.62	95.43	4.00	0.00	0.29	0.29	84,286	733
nasa7	6,128	3.08	79.29	81.34	6.34	0.41	5.95	5.95	83,867	706
ora	6,036	7.52	53.24	69.85	10.65	0.00	9.75	9.75	70,604	668
spice	16,148	12.57	71.63	91.56	3.73	0.16	2.28	2.28	138,312	815
su2cor	4,777	4.36	73.07	76.42	9.02	0.71	6.92	6.92	93,668	711
swm256	11,037	1.65	98.42	99.63	0.15	0.07	0.08	0.08	73,412	677
tomcatv	900	3.36	99.28	99.86	0.05	0.02	0.03	0.03	65,625	617
wave5	3,555	5.71	61.79	76.68	5.92	0.74	8.33	8.33	106,978	762
alvinn	5,241	9.09	97.77	98.30	0.40	0.02	0.64	0.64	17,811	212
compress	93	13.91	68.25	88.51	7.59	0.00	1.95	1.95	13,144	149
ear	17,006	8.10	90.13	61.37	3.71	0.05	17.42	17.46	25,079	290
eqntott	1,811	11.54	63.03	93.47	1.90	1.70	0.70	2.24	19,172	212
espresso	513	17.11	61.90	93.25	1.88	0.20	2.29	2.39	60,674	551
gcc	144	15.97	59.42	78.85	5.75	2.86	6.04	6.49	186,066	1,651
li	1,355	17.67	47.30	63.94	7.74	2.24	12.92	13.16	33,235	551
sc	1,450	20.93	64.34	85.96	2.62	0.98	5.18	5.26	59,291	512
cfront	17	13.37	53.14	76.02	5.62	2.59	7.89	7.89	225,064	981
db++	86	17.56	56.86	54.43	2.03	15.04	6.77	21.73	20,784	329
groff	57	17.51	49.20	66.22	10.22	3.17	9.09	11.30	121,191	1,756
idl	21	19.61	46.70	50.00	7.55	12.31	9.07	21.07	79,381	1,459
lic	6	16.79	52.26	65.76	8.78	0.22	12.58	12.66	384,058	5,333
porky	164	19.76	60.48	55.34	2.82	3.14	17.92	20.78	216,678	3,704

Table 1: Measured Attributes of the Traced Programs. Dynamic information is recorded from a particular execution of the application, while static information is a property of the program binary.

mispredicted branches (%MpB), but it is often difficult to understand how these metrics influence processor performance. Yeh & Patt [41] defined the *branch execution penalty* to be:

$$\text{BEP} = \frac{\text{\%MfB} \times \text{misfetch penalty} + \text{\%MpB} \times \text{misprediction penalty}}{100},$$

which reflects the average penalty suffered by a branch due to misfetch and misprediction. A BEP of 0.5 means that, on average, each branch takes an extra half cycle to execute; values close to zero are desirable. We use this metric to provide a more intuitive understanding of how the two penalties interact. However, using the BEP binds us to a specific misfetch and misprediction penalty, so we also report the %MfB, %MpB along with the BEP. For the results in this paper, we use a one cycle misfetch penalty and a four cycle mispredict penalty.

In the Precomputed-Branch architecture, some branches are changed to indirect branches. We assume this is done by loading the value from memory and performing an indirect jump. A single-cycle penalty for loading the branch destination is included in the BEP for the Precomputed-Branch architecture. Thus, we extend the BEP model to be:

$$\text{BEP} = \frac{\text{\%MfB} \times \text{misfetch penalty} + \text{\%MpB} \times \text{misprediction penalty} + \text{\%IIB} \times \text{extra indirect branch penalty}}{100},$$

where $\%IIB$ is the percent *increase in indirect branches*. This is the percentage of PC-relative branches converted to indirect jumps, expressed as an average cost over all branches executed for the Precomputed-Branch architecture. The %IIB is determined when the program is partitioned into branch spaces, as described in the next section. For the BTB architecture results, we assumed a 21-bit branch displacement, so %IIB is always zero.

5 Partitioning Programs Into Branch Spaces

In this section, we show the overhead introduced by the Precomputed-Branch architecture when the instruction set only permits small displacements. We use static methods to partition the program into multiple branch spaces, and then compare the benefits of partitioning using information from prior execution. In practice, most architectures provide branch instructions intended to be used within a single procedure and different branch instructions used to transfer control to other procedures. Often, the displacements in these instructions are different sizes; for example, the MIPS architecture uses 26-bit displacements for procedure calls, and 16-bit displacements for conditional branches. By comparison, the DEC Alpha uses 21-bit displacements for all branches.

When using PC-relative addresses, procedures can be placed anywhere in virtual memory. By comparison, the Precomputed-Branch architecture places more restrictions on procedure placement. In a Precomputed-Branch architecture, a branch located at address X is located in a particular Z-bit branch space specified by $S(X, Z) = \lfloor \frac{X}{2^Z} \rfloor$. Branches can only reach destinations in the same branch space; thus, to branch from X to Y, we must insure that $S(X, Z) = S(Y, Z)$ in order to use a pre-computed branch. If $S(X, Z) \neq S(Y, Z)$ then an indirect jump must be used to branch between spaces. Branches within a procedure and between procedures must be able to reach their destinations. To accomplish this, we reorganize the program trying to insure that all branches and their destination are in the same branch space. If this restriction can not be enforced, the branch is converted into an indirect jump that can span across branch spaces.

In this paper, we partition programs into branch spaces of three different sizes: 14, 16 and 21-bit branch spaces. We chose the 16 and 21-bit branch spaces because they reflect the branch displacement in existing microprocessors, and it is easy to argue that branch spaces of this size are easy to implement. However, all of our sample programs fit within a single 21-bit branch space, and most of the programs fit in two 16-bit branch spaces. Therefore, we also partitioned programs into a 14-bit branch space to give some insight of the overheads that might be encountered by larger programs. We further stipulated that procedures are not spilt across branch spaces; only procedure calls will span branch spaces. Therefore, all intra-procedural branches will be compiled as pre-computed branches.

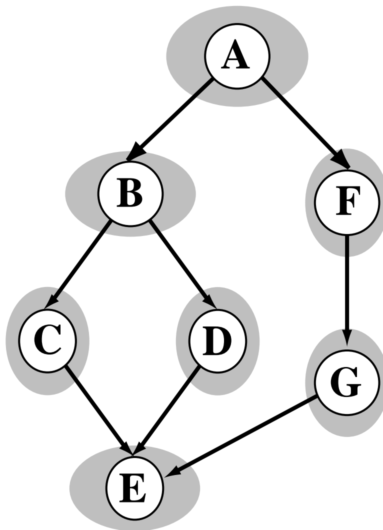
There are myriad ways to partition programs, and a number of alternatives have been examined in the effort to reduce page faults [1, 2, 15, 13], and instruction cache conflicts [16, 25, 30]. The goals of our study are different than these other studies; we are more interested in reducing the number of indirect jumps than reducing cache conflicts and paging. None the less, the best performing algorithm we examined (MaxCut) for code partitioning is very similar to the greedy layout algorithm of Pettis and Hansen [30]. Therefore, partitioning the program into branch spaces using the MaxCut algorithm would result in a layout that also reduces cache conflicts and paging.

We used two metrics to compare the program partitioning heuristics. The first is the additional amount of space needed by the program due to wasted space at the end of virtual memory pages. The second metric is the percent of dynamic branches that cross branch spaces. We considered a number of partitioning algorithms. Some methods use profiles, or information about previous executions of the program. Many optimizations require such information, either from previous executions of the program or from estimates using static analysis [36, 39]. We examined depth-first, breadth-first, pre-order, post-order, greedy and max-cut partitioning algorithms. Most of the methods had similar performance, and we present the performance of three of these algorithms.

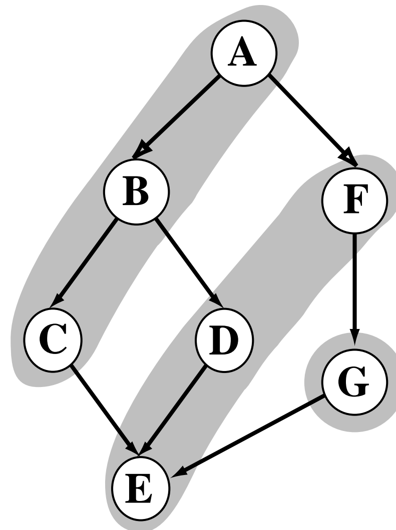
The partitioning algorithms are illustrated in Figure 4. In this example, we assume that each branch space can hold three procedures. Branch spaces are indicated by the darkened regions. The costs shown in Figure 4 reflect the number of branches that cross branch spaces for a particular execution of the program. There are a total of 39 procedure calls in the example, and each partitioning algorithm attempts to reduce the number of procedure calls that span branch spaces. In addition, the profile-driven partitioning heuristics use the procedure call weights while partitioning the procedures.

The Separate method, shown in Figure 4(a), partitions each procedure into a different branch space, and illustrates the worst-case performance one could encounter. In the Preorder partitioning, nodes are added to a mapping list in a pre-order traversal without using profile information. That list is then partitioned into branch spaces. A similar technique is used in the Depth-First Profile method; a depth-first search orders the nodes, always visiting the out-going edge with the highest call frequency. This Depth-First Profile algorithm is very similar to the procedure layout algorithm proposed by Hwu and Chang [25]. The MaxCut partitioning uses a greedy max-cut algorithm to partition the graph using the call frequency to guide the partitioning. This algorithm is very similar to the greedy approach for procedure mapping proposed by Pettis and Hansen [30].

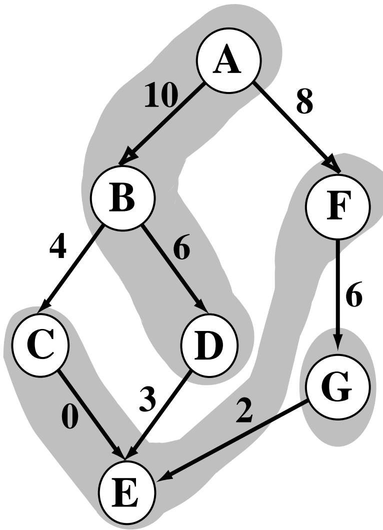
The MaxCut partitioning algorithm processes the edges in the call graph, starting with the most frequently executed call edge and ending with the least executed edge. For this algorithm we group procedures together into *branch-spaces* until each branch-space is full. Therefore, in the final partition each branch-space contains a group of procedures, where each procedure will use pre-computed branches to call all the other procedures in that branch-space and indirect jumps to call procedures not in that branch-space. Once a



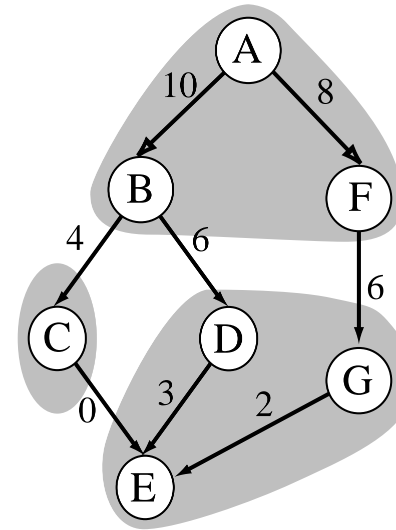
(a) Separate - Each procedure forms a branch space, and no profile information is used. The execution cost is 39, using the weights shown below.



(b) Preorder - The traversal order is {A, B, C, E, D, F, G}, and no profile information is used. The execution cost is 25, using the weights shown below.



(c) Depth-First Profile - The traversal order is {A, B, D, C, E, F, G}. The execution cost is 21, using the weights shown.



(d) MaxCut - The order branch-spaces were formed was {A, B, F, D, E, G, C}. The execution cost is 16, using the weights shown.

Figure 4: Partitioning A Simple Call Graph. In this example we assume that at most three procedures can fit into one branch space. In these graphs the nodes labeled with letters are procedures, the edges are procedure calls, the numbers indicate the number of procedure calls, and the shaded areas represent the branch spaces the procedures are partitioned into for each algorithm. The cost shown for each partitioning is the number of procedure calls converted to indirect jumps because they span across different branch spaces.

branch-space is full we do not add any more procedures to that branch-space when performing the partition. A branch-space is considered full when the total number of instructions of all of the procedures in the branch-space is equal to the target range of the branch architecture's explicit displacement. For example, for a branch architecture that has 16-bit explicit displacement, a branch-space can at max hold 2^{16} or 65,536 instructions, and is considered full when it reaches that limit. When an edge is processed for the MaxCut algorithm, it connects two procedures and these two procedures could be in one of the following three states: (1) both procedures are unprocessed, so neither of the procedures has been added to a branch-space partition, (2) one procedure is already in a branch-space and the other has not yet been processed, or (3) both procedures are already processed and in a branch-space. For the first case, both of the procedures connected by the edge are grouped together into a new branch-space. For the second case, the unprocessed procedure is added to the branch-space of the already processed procedure, only if adding the procedure keeps the size of the branch-space less than the branch architecture's target range. For the third and final case, if the two procedures for the edge being processed are in the same branch-space then nothing needs to be done. If they are in different branch-spaces then the two branch spaces are merged together if their total size fits within the branch architecture's target range. After all the edges have been processed, we are left with several disjoint branch-spaces, many of which are smaller than the architectures target range. These final branch-spaces are merged together as long as the merge does not create a new branch-space larger than the target range. After this is completed, the branch-spaces are then layed out from the most frequently executed to the least frequently executed, leaving fluff at the end of the branch-space if the branch-space was not completely full.

The example call graph and call frequencies in Figure 4 show that the Preorder partitioning algorithm, which does not use a profile, has 5 procedure call edges that span across branch spaces and these account for 25 executed procedure calls uses the example's call frequencies. In comparison, the profile based MaxCut algorithm has 4 procedure call edges that span across different branch spaces with an execution frequency cost of 16 procedure calls. This example shows that the MaxCut algorithm can substantially decrease the branches that span across the branch space over the Preorder partitioning algorithm.

5.1 Performance of Program Partitioning Heuristics

Table 2 summarizes the performance of the partitioning heuristics for a 14-bit and 16-bit branch space. [1] The Table is broken into four major columns, showing the performance of each method. For each method, the first sub-column (Seg) shows the number of branch spaces introduced by the partitioning, and the number of additional program pages needed by the partitioned program over the original program. We assumed an 8KByte page size. The next sub-column (%IIB) shows the increase in indirect function calls introduced by this partitioning. This term is used when computing the Branch Execution Penalty. Average results are shown for the two branch spaces for the benchmarks broken down into 5 categories: the 10 Perfect-Club benchmarks, the 11 SPECfp92 benchmarks, the 8 SPECint92 benchmarks, the 6 C++ programs, and the overall average of all 35 programs. For example, Table 2 shows the Overall Average for all the programs partitioned using the Preorder method with a 14-bit branch space; 7 branch spaces are used, 1 additional 8KByte memory page is needed, and 2.29% of the branches are converted to indirect jumps. The %IIB value is averaged over all branches to simplify the calculation of the BEP, and provides insight to the overhead of

[1] The detailed results for partitioning each program into 14-bit and 16-bit branch spaces are included in the appendix.

Branch Space	Programs	Separate		Preorder		Prof Depth		Max Cut	
		Seg	%IIB	Seg	%IIB	Seg	%IIB	Seg	%IIB
14-Bit Branches	Perfect-Club	743	4.34	7/1	2.86	7/1	0.54	7/0	0.47
	SPECfp92	709	3.96	6/1	1.00	6/2	0.98	6/0	0.38
	SPECint92	516	5.67	4/1	0.59	4/0	0.49	4/0	0.37
	C++	2260	10.46	12/4	5.97	11/2	3.85	11/0	3.35
	Overall Avg	940	5.57	7/1	2.29	7/1	1.23	7/0	0.91
16-Bit Branches	Perfect-Club	743	4.34	2/0	0.49	2/0	0.36	2/0	0.00
	SPECfp92	709	3.96	2/0	0.16	2/0	0.34	2/0	0.00
	SPECint92	516	5.67	1/0	0.13	1/0	0.11	1/0	0.11
	C++	2260	10.46	3/0	2.34	3/0	1.88	3/0	1.66
	Overall Avg	940	5.57	2/0	0.62	2/0	0.56	2/0	0.31

Table 2: Summary of Performance for Branch Partitioning Heuristics. The segment column shows the number of branch spaces (segments) and the extra virtual memory pages needed by that partitioning. The %IIB column shows the number of branches converted to indirect branches required by that partitioning, averaged over all branches in the program.

partitioning on total branch execution. The same program execution was used to partition and then assess the profile-directed partitioning methods; the performance of other executions may differ.

Table 2 shows that the algorithm used to partition the programs into a 14-bit branch space has a reasonable impact on program size and the number of indirect jumps. This effect is largest for the C++ program average, where the Preorder partition results in an increase in memory usage of four 8K pages, and 6% of the branches are converted to indirect jumps. When adding profile information, the MaxCut partition requires no additional memory usage for the C++ programs, and the percent of branches that are converted to indirect jumps is reduced by almost a half down to 3.3%. When using a 16-bit branch space there was little effect on program size or the number of indirect jumps for the benchmarks examined. For this branch space, the C++ program average again showed the biggest improvement, with the percent increase in indirect branches (%IIB) reduced from 2.3%, when using the Preorder algorithm, down to 1.7%, when using the MaxCut algorithm. As mentioned, all of the programs we examined fit into a single 21-bit branch space when using the Precomputed-Branch architecture, so there is no difference between the different partitioning algorithms for a 21-bit branch space.

When comparing branch target buffer architectures to the Precomputed-Branch architecture, we will only consider the Preorder and MaxCut partitioning, and show the results for 14, 16 and 21-bit explicit branch displacements. The Preorder partitioning provides the lowest branch cost for those methods we examined that did not used profile information, while the MaxCut method provided the best profile-driven partition.

6 Branch Architecture Performance Comparison

In this section we compare the performance of the branch target buffer (BTB) architecture to the Precomputed-Branch architecture. We use the branch execution penalty to compare performance, but also report the misfetch and misprediction penalty.

Both architectures used a 4096-entry decoupled 2-level pattern history table, using the extension proposed by McFarling [23], where the program counter is XOR-ed with the global history register to form an index into the PHT. In the BTB architecture, we simulated branch target buffers with 4, 16, 32, 64, 128, 256, 512 and an infinite number of entries. In each case, the BTB was organized as a 4-way associative BTB. The Precomputed-Branch architecture stores the pre-computed branch destination as part of the instruction for all branches except indirect jumps, with no additional overhead in the instruction cache. We used a small 4-way associative indirect jump buffer (IJB) with the Precomputed-Branch architecture, containing either 0, 4, 16 or 32 entries, to predict the destination of indirect branches. We also simulated a direct-mapped IJB for the same configurations.

6.1 Comparing the BTB and Precomputed-Branch Architectures

Figure 5 summarizes the branch execution penalty for the different partitioning methods, branch space sizes and architectures. The Preorder-14 column uses the Preorder partitioning heuristic with 14-bit branch spaces. The MaxCut-14, Preorder-16 and MaxCut-16 uses the corresponding partitioning and branch sizes. The Optimal-21 partitioning uses a 21-bit branch space. With this branch space size, each program in our benchmark suite occupied a single branch space, and there is no difference between the Preorder and MaxCut partitioning. The Precomputed-Branch architecture has no misfetch penalty because the destination for all direct branches is immediately known. The BTB-Misfetch=1 column shows the performance for the decoupled BTB architecture with a single cycle misfetch penalty. A larger misfetch penalty would increase the BEP for the BTB-based architecture, but not the Precomputed-Branch architecture.

Table 3 provides more details about the terms used to compute the BEP. We show the average percent of misfetch and mispredicted branches. For the Precomputed-Branch architecture, we also show %IIB, the cost of converting procedure calls that span branch spaces into indirect branches. A misfetched branch occurs because the processor does not know the destination address for a taken branch. The BTB architecture misfetches because the computed destination is not found in the BTB, while the Precomputed-Branch never misfetches because the pre-computed destination is stored in the instruction. A mispredicted branch occurs either because the outcome of a conditional branch is mispredicted, or the destination of a procedure return or indirect branch is mispredicted.

6.2 Performance Analysis

Figure 5 shows that the Precomputed-Branch architecture with no additional prediction for indirect branches is comparable in performance to the BTB architecture with 16 to 64 entries. The Precomputed-Branch configurations with 14-bit and 16-bit branch spaces fairs worse than the 21-bit design, because a larger number of branches must span branch spaces (increasing %IIB) and because those branches are converted to indirect branches which may be mispredicted (increasing %MpB). Partitioning using profiles, as in the MaxCut partitioning, reduces the penalty for small branch spaces. It should be noted that the BTB results shown in this paper use a 21-bit displacement for the branch instructions. If the programs were studied for the BTB architecture with a smaller displacement, then possibly some PC-relative branches would need to be converted to indirect jumps just as in the Precomputed-Branch architecture design. Therefore, for a fair comparison, the BTB results should be compared directly to the 21-bit explicit displacement Precomputed-Branch architecture results (Optimal-21).

Architecture	BTB Size	%MfB	%MpB	%IIB	BEP
BTB	4	22.88	4.85	0.0	0.42
BTB	16	12.64	4.66	0.0	0.31
BTB	32	8.62	4.43	0.0	0.26
BTB	64	4.57	4.37	0.0	0.22
BTB	128	2.57	4.31	0.0	0.20
BTB	256	1.08	4.27	0.0	0.18
BTB	512	0.44	4.25	0.0	0.17
BTB	Infinite	0.02	4.23	0.0	0.17
Preorder-14	0	0.0	7.75	2.29	0.31
Preorder-14	4	0.0	5.69	2.29	0.23
Preorder-14	16	0.0	4.78	2.29	0.19
Preorder-14	32	0.0	4.60	2.29	0.18
MaxCut-14	0	0.0	6.37	0.91	0.24
MaxCut-14	4	0.0	5.37	0.91	0.21
MaxCut-14	16	0.0	4.58	0.91	0.18
MaxCut-14	32	0.0	4.46	0.91	0.18
Preorder-16	0	0.0	6.07	0.62	0.26
Preorder-16	4	0.0	5.13	0.62	0.22
Preorder-16	16	0.0	4.53	0.62	0.18
Preorder-16	32	0.0	4.46	0.62	0.18
MaxCut-16	0	0.0	5.77	0.31	0.23
MaxCut-16	4	0.0	4.97	0.31	0.20
MaxCut-16	16	0.0	4.42	0.31	0.18
MaxCut-16	32	0.0	4.35	0.31	0.17
Optimal-21	0	0.0	5.45	0.0	0.22
Optimal-21	4	0.0	4.69	0.0	0.19
Optimal-21	16	0.0	4.29	0.0	0.17
Optimal-21	32	0.0	4.26	0.0	0.17

Table 3: Summary of Performance Information From Trace Driven Simulations. The branch execution penalty is computed with a 4-cycle branch misprediction penalty, a 1-cycle misfetch penalty and a 1-cycle penalty for extra indirect branches. The values shown are arithmetic means over all programs.

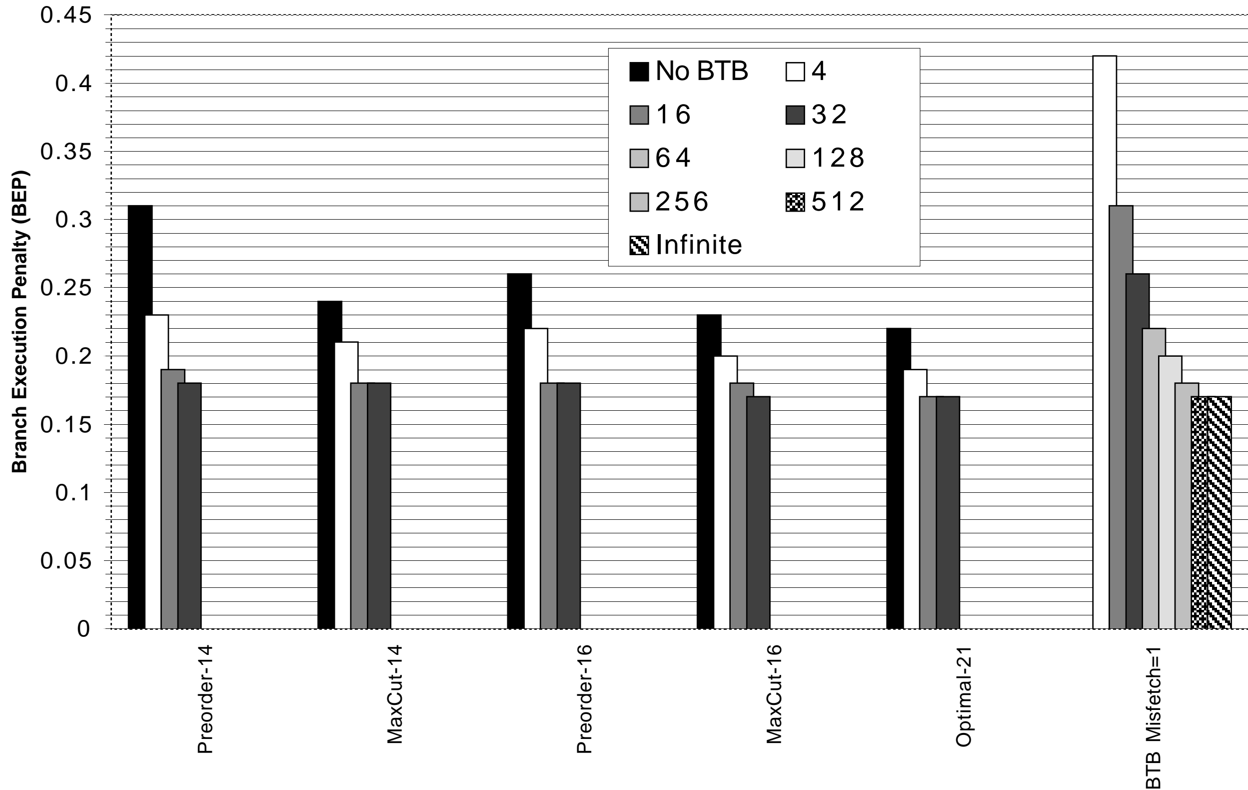


Figure 5: Branch Execution Penalty. The Branch Execution Penalty is computed with a 4-cycle branch misprediction penalty, a 1-cycle misfetch penalty and a 1-cycle penalty for converted indirect branches. Results are shown for a 14-bit, 16-bit, and 21-bit branch space.

Adding a very small indirect jump buffer for indirect branches to the Precomputed-Branch architecture provides as much benefit as profile-based partitioning, and reduces the BEP for each design. With a small 16 or 32 entry IJB, the Precomputed-Branch design has the same BEP as a processor using an infinitely large BTB. Most of the performance gain is evident even when a small number of IJB entries are used. In the Precomputed-Branch design, the IJB is only used to predict indirect jumps, while the design using a BTB must use the BTB to avoid misfetching for all PC-relative branches and to predict the destination for indirect jumps.

6.2.1 Impact on Chip Area

To evaluate the area implementation costs for these architectures we used the register bit equivalent (RBE) cost model for on-chip memories proposed by Mulder *et al.* [27], where one RBE equals the area cost of a single bit storage cell. Figure 6 graphs the performance for the Precomputed-Branch architecture and the BTB design showing the branch execution penalty on the Y-axis and the register bit equivalent chip area costs on the X-axis (the lower the RBE cost the better). Results are shown for a 4, 16, 32, 64, 128, 256, and 512 entry BTB for a processor which has a 32-bit and 64-bit address space. The Precomputed-Branch results are shown for an architecture with instructions that have a 21-bit explicit displacement, with no IJB,

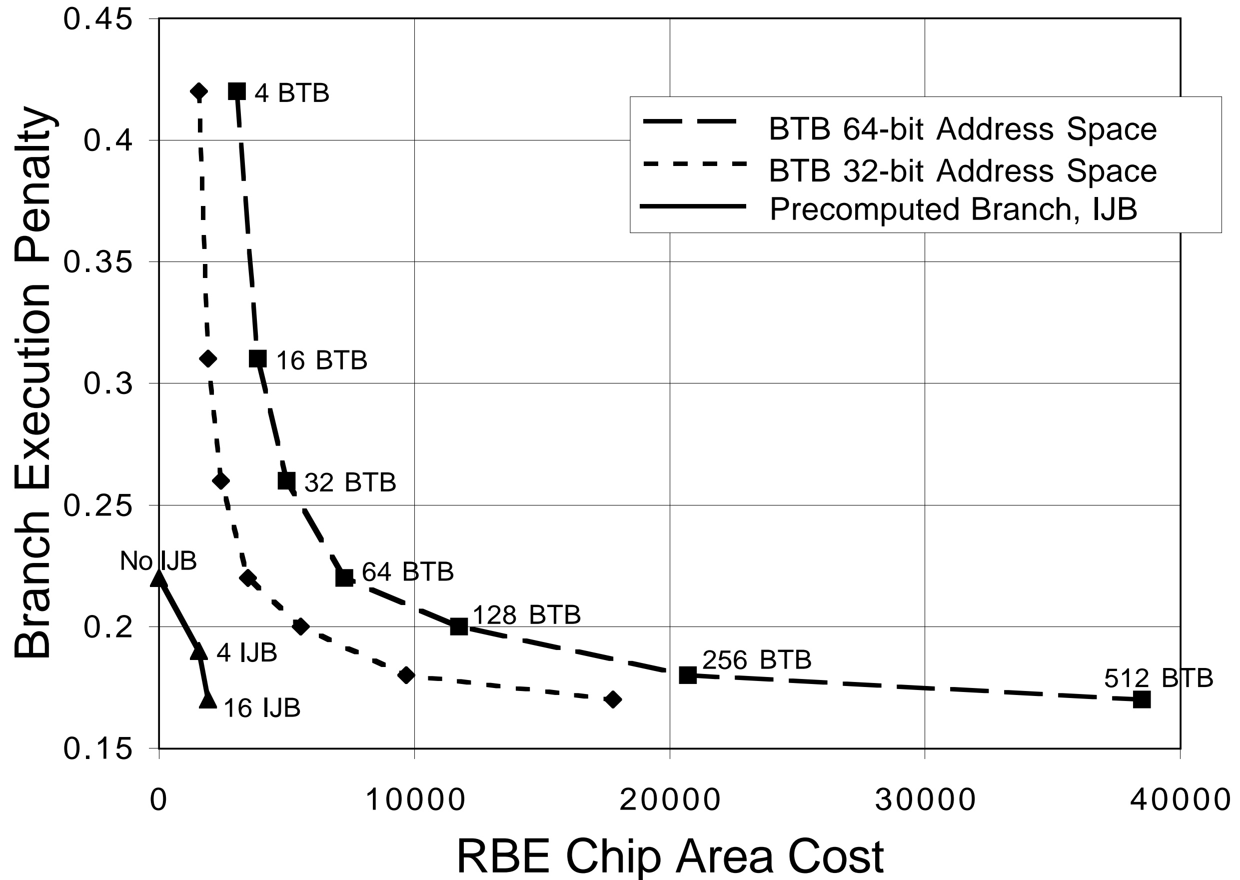


Figure 6: Branch Execution Penalty and Register Bit Equivalent Chip Area Costs. Results for a 4, 16, 32, 64, 128, 256, and 512 entry BTB are shown for an architecture with a 32-bit and 64-bit address space. This is compared to the Precomputed-Branch performance for a 21-bit explicit displacement with no IJB, and a 4 and 16 entry IJB.

and a 4 and 16 entry IJB. For example, in Figure 6 the 16 entry IJB Precomputed-Branch architecture has a register bit equivalent cost of 1,900 rbe, with a branch execution penalty of 0.17.

Figure 6 shows an important design difference between the Precomputed-Branch architecture and the BTB – chip area cost. The BTB's chip area cost is dependent on the size of the processor's address space. As future processors change from a 32-bit address space to a 64-bit address space, the chip area cost for a BTB will significantly increase. In comparison, the increased chip area cost for the Precomputed-Branch architecture is small and comes from the very small IJB needed to predict indirect jumps.

The RBE cost for a direct mapped 512 entry BTB is 19,000 rbe for a 32-bit address space. In comparison, an 8KByte direct mapped instruction cache with 32 byte lines has an RBE cost of 44,000 rbe. Therefore, the hardware cost of the BTB is around 43% of the cost of an 8K direct mapped instruction cache. Since the main purpose of the BTB is to eliminate misfetch penalties by providing taken target addresses, the BTB is a costly mechanism in comparison to the reduction is memory latency that an instruction cache provides, when comparing the hardware costs for these two mechanisms. This Figure shows that the Precomputed-Branch architecture is a very attractive alternative to the BTB design especially for future 64-bit processors.

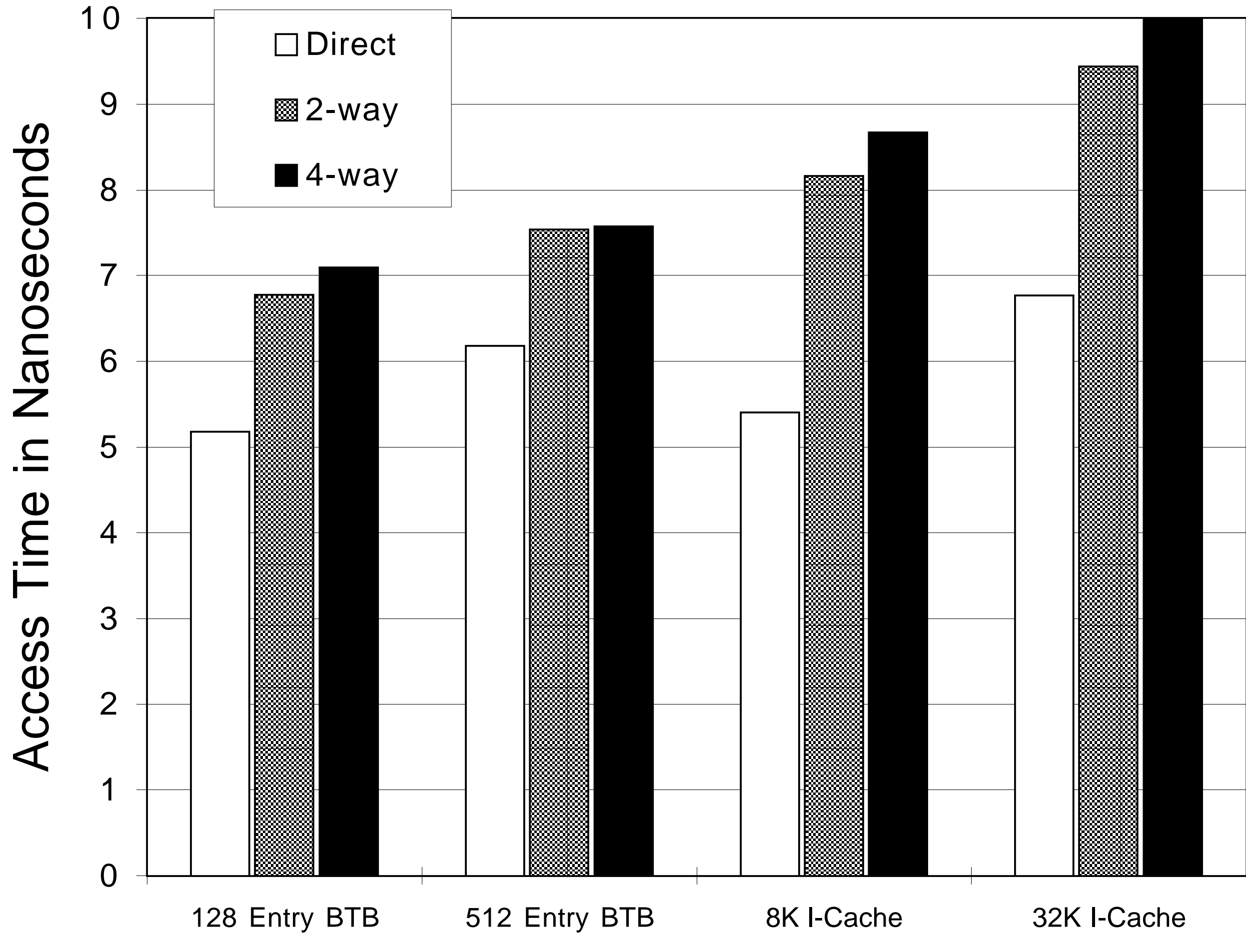


Figure 7: BTB and Instruction Cache Access Time. Access time is in nanoseconds for a direct mapped, 2-way and 4-way associative caches. Access times are shown for a 128 and 512 entry BTB and a 8K and 32K instruction cache with 32 byte lines.

6.2.2 Impact on Cycle Time

The access time for a cache (or a BTB) depends both on the size and the associativity of the design [17, 31]. Recall that the branch target buffer requires considerable resources, and is organized as a 4-way associative cache, while the Precomputed-Branch architecture uses information recorded in the instructions. Figure 7 shows the access time of the BTB in comparison to the access time for an instruction cache using an accurate timing analysis tool by Wilton and Jouppi [38]. The Figure shows that the access time of a 4-way associative 128 entry BTB is larger than the access time for a direct mapped 8K and 32K instruction cache with 32 byte lines. The reason for this access time difference is that for a direct mapped cache no mux driver is needed, and the data contents can be sent ahead to the next pipeline stage before the full tag comparison finishes. For associative caches, the associative tag comparison and selection can significantly slow down the cache access time. Also notice in Figure 7, that the access time of the direct mapped 512 entry BTB is larger than the direct mapped 8KByte instruction cache. This is because of the slower tag array organization resulting from the 512 tags in the BTB, compared to the 256 tags needed for the 8KByte cache. Notice that the 512 entry BTB has a smaller access time than the 8KByte cache when comparing 2-way and 4-way associative

		IJB Entries			
		0	4	16	32
Direct Mapped IJB	BEP	0.22	0.19	0.18	0.17
	%MpB	5.51	4.67	4.40	4.35
4-way Associative IJB	BEP	0.22	0.19	0.17	0.17
	%MpB	5.45	4.69	4.29	4.26

Table 4: A comparison of the BEP and %MpB for the 21-bit branch space partitioning with direct and associative IJB designs.

organizations. This is because of larger and slower mux driver needed to drive the 32-byte line of the cache compared to the 4-byte line of the BTB.

Despite the advantages of direct mapped caches, many BTB designs, such as the BTB used in the Intel Pentium and PentiumPro and those proposed by Yeh *et al* [41] use a large, multi-associative BTBs to reduce the misfetch penalty. Though, for processors like the DEC Alpha 21064 and 21164, which have a direct mapped first level instruction cache, an associative BTB design is not practical because of the access times shown in Figure 7. Since the instruction fetch cycle often limits processor performance, designs using a large, associative BTBs may lengthen the cycle time, affecting the performance of the entire processor. In [18], the designers of the TFP (MIPS R8000) microprocessor stated:

> *We evaluated several well-known branch prediction algorithms for layout size, speed, and prediction accuracy. The most critical factor affecting area was the infrastructure required to support a custom block: power ring and power straps to the ring, and global routing between the branch prediction cache and its control logic. Speed was a problem with tag comparisons for those schemes that are associative. Accordingly we chose a simple direct-mapped, one-bit prediction scheme which can be implemented entirely with a single-ported RAM.*

The access times in Figure 7 show that the BTB can affect the cycle time of the processor. In comparison, the Precomputed-Branch design would have only a very small 16 entry IJB which would not affect the cycle time in comparison to the instruction cache access time. The indirect jump buffers simulated in Figures 5 and 6 were 4-way associative as were the BTB designs. Table 4 compares the performance of the 21-bit branch space partitioning with a direct-mapped and associative IJB; there is little difference in performance.

6.2.3 Impact of Large Applications

In general, the BTB branch architecture has worse performance then the Precomputed-Branch architecture for the large programs in the benchmark suite, although this is not indicated in the mean values we show. In part, this is a reflection of the application mix we used for benchmarking. Although applications such as the Perfect Club and SPEC suite have the advantage of being well known and understood, they are dominated by a small number of heavily executed branches, primarily in loops. By comparison, benchmarks such as `cfront` and `groff` have more branches, and better illustrate the problems of fixed capacity mechanisms such as BTB's. We feel these programs better illustrate the performance of branch architectures. These characteristics are shown in Table 5. Table 5 shows the *branch quantiles* for each program, and the total

Program	Q-25	Q-50	Q-60	Q-70	Q-80	Q-90	Q-95	Q-99	Q-100	Static-All
APS	20	50	74	113	177	322	425	631	2705	17460
CSS	11	50	80	128	190	293	356	650	3293	19904
LGS	4	12	18	25	39	65	92	147	1972	14910
LWS	2	4	7	11	16	28	39	57	1686	14580
NAS	3	7	11	16	24	66	121	200	2574	15594
OCS	1	3	4	7	18	49	83	219	2215	14524
SDS	1	9	13	21	30	44	74	204	2475	15298
TFS	6	16	23	34	60	152	267	571	2368	14823
TIS	2	8	11	17	23	31	36	66	1201	13049
WSS	12	49	90	143	217	327	413	641	2547	15334
doduc	2	6	16	40	116	237	301	403	1965	14159
fpppp	6	11	18	27	39	60	88	130	1029	13039
hydro2d	7	15	25	36	56	77	113	234	2177	14253
mdljsp2	4	7	8	10	12	15	20	30	1472	14416
nasa7	3	10	15	23	42	78	133	381	1594	13680
ora	3	6	8	10	12	14	17	24	897	12431
spice	1	3	6	12	23	50	90	155	2718	19060
su2cor	4	10	13	17	24	37	48	80	2114	14658
swm256	1	2	2	2	2	3	3	15	1146	12745
tomcatv	2	3	3	4	4	5	7	7	723	11515
wave5	7	22	31	46	71	111	179	355	1789	15378
alvinn	1	2	2	2	2	2	6	140	570	3107
compress	2	5	6	8	10	15	18	20	339	2236
ear	2	3	3	4	5	7	9	43	844	3968
eqntott	1	2	2	2	5	26	58	94	740	3205
espresso	15	48	69	98	133	186	289	686	2843	9728
gcc	73	326	540	852	1308	2180	3182	5341	11781	32250
li	8	28	38	49	66	98	149	224	1082	5827
sc	4	12	26	44	63	114	199	518	2492	9983
cfront	29	99	169	309	615	1354	2271	4592	9228	37142
db++	3	9	18	33	68	134	193	256	829	3569
groff	44	168	237	326	449	697	990	1819	5675	19534
idl	6	15	19	22	27	64	127	398	2850	12433
lic	29	99	156	266	449	752	1088	2074	4108	54222
porky	3	16	40	65	124	358	649	1714	6258	29656
Overall Avg	9	32	51	80	129	230	346	660	2579	15362

Table 5: Branch quantiles, showing the contribution of individual branch instructions to the branch activity in each program. The value of a given quantile entry shows the number of individual branch instructions that contribute to a given fraction of the branching activity in a program.

Configuration	BTB/IJB	cfront		gcc		groff		lic	
		%MfB	%MpB	%MfB	%MpB	%MfB	%MpB	%MfB	%MpB
BTB	256	12.52	13.79	6.91	12.39	4.95	5.35	4.80	6.75
BTB	Infinite	0.25	13.61	0.03	12.30	0.04	5.13	0.25	6.74
Optimal-21	0	0.00	14.39	0.00	13.42	0.00	7.51	0.00	6.90
Optimal-21	4	0.00	13.93	0.00	12.48	0.00	6.34	0.00	6.77
Optimal-21	16	0.00	13.75	0.00	12.33	0.00	5.45	0.00	6.74
Optimal-21	32	0.00	13.66	0.00	12.32	0.00	5.31	0.00	6.74

Table 6: Comparison of Misfetch and Mispredict Penalties for Programs with Many Branches. Both the BTB and IJB are 4-way associative.

number of branches executed in that program. Each branch quantile shows the number of static branch sites that contribute a given amount to the number of dynamic branches during execution. For example, in the 'APS' program, the Q-90 value indicates that the 322 branch instructions in the program constitute 90% of the branches executed in the program. The Q-100 value shows the total number of branch sites executed in the program, and the Static-All value shows the total number of static branches in the program.

Some programs, such as `compress` and `su2cor`, have a high misprediction rate with few branches, because they contain a few conditional branches that are simply hard to predict. Other programs, such as `cfront` and `gcc` contain a great number of conditional branches, and many of those branches are difficult to predict. The prediction accuracy for these programs can be improved by using larger pattern history tables, or by various compiler transformations [6, 25, 43]. One advantage of the Precomputed-Branch architecture, particularly for larger programs representative of actual applications, is that it is less susceptible to capacity misses for fetch prediction.

Programs such as `cfront`, `gcc`, `lic` and `groff` contain a large number of branches, and the Precomputed-Branch architecture performs very well for for these programs. Table 6, which shows the %MfB and %MpB, shows why. Each program has a high misprediction rate in both architectures, reflecting the unpredictability of the conditional branches in these applications. The BTB architecture has a slightly lower %MpB than the Precomputed-Branch architecture if no IJB entries are used, because the BTB architecture can predict indirect jumps. However, the BTB architecture must also use the BTB to avoid instruction misfetch penalties, and even a 256-entry BTB has a considerable number of misfetches for these large programs.

The C++ programs can benefit greatly from the addition of a very small indirect jump buffer. For example, the %MpB for the `db++` program drops from 15.71 to 0.78 with the addition of a four-entry IJB. This program contains a large number of indirect jumps, as shown by the detailed program information in Table 1, but those indirect jumps are fairly predictable. The predictability of indirect jumps in C++ programs was shown in an earlier study [7], and has been demonstrated for many C++ programs. Tables 9 and 10, in the appendix, show the %MpB for all configurations of the Precomputed-Branch architecture, using both partitioning algorithms.

7 Practical Concerns

In the past, segment architectures have been greeted with less than overwhelming enthusiasm, due to limited segment sizes. However, the Precomputed-Branch architecture has a single instruction address space with branch instructions that can only access a portion of that address space. A concern with an explicit displacement encoding is how to implement code relocation, which is important for shared program libraries. For the Precomputed-Branch architecture, consider using an architecture such as the DEC Alpha AXP, which uses a 21-bit branch displacement for word-aligned instructions in a 64-bit instruction address space. The instruction space is broken into $2^{64-21+2}$, or $\approx$ 2 trillion branch spaces of 8MBytes each. Branches within each 8MByte branch space use an explicit displacement (pre-computed branch). Each segment can be relocated to $\approx$ 2 trillion different locations without modification, and all branches within a given branch space are relative to that branch space. Furthermore, such large 8MByte branch spaces would address almost all programs we have encountered. With $\approx$ 2 trillion different branch spaces to choose from for relocation, dynamically performing relocation for a program or a shared library would not be a problem for the Precomputed-Branch architecture.

7.1 Non-relative Branches in a Relative World

The Precomputed-Branch architecture performs most of its branch computation at link or compile time. Traditional relative branches perform the branch computation during instruction issue, where branch target buffers can be used to cache this information. The primary objection to using non-relative branches is that most instruction sets already use relative branches, and the Precomputed-Branch architecture requires a change to the instruction set architecture.

It is also possible to pre-compute the branch destination when instructions are entered into the instruction cache. Instructions in many architectures are partially decoded when fetched into the cache, simplifying instruction dispatch and scheduling. The destination for a relative branch instruction can also be computed during instruction fetch. This is shown diagrammatically in Figure 8. The lower order bits of the branch instruction's PC-relative target address is computed up to the carry-bit as the instruction is brought into the cache; if the carry-bit is set, the branch destination is not in the current branch space. If the branch instruction branches to a destination within the same branch space, the decoded instruction type is set to indicate that the destination contains a pre-computed branch, and the instruction stored in the instruction cache is changed to contain the pre-computed explicit displacement. If the destination is not in the same branch space, the branch will have to use an indirect jump, and the instruction type is marked to indicate that the IJB should be used to predict the branch destination. When fetching a branch instruction with the pre-computed branch type bit set, the target address for the next fetch is calculated as previously described for the Precomputed-Branch architecture, where the upper bits of the PC are concatenated with the pre-computed branch displacement. For a branch instruction that is marked to use the IJB, the first time the instruction is executed the proper destination is computed in the decode stage and is used to initialize the IJB. Then on subsequent accesses to that instruction, the target address stored in the IJB will be used in the next instruction fetch to eliminate the misfetch penalty.

In such an architecture, the partitioning discussed in §5 reduces the number of branches that span branch spaces. As in the Precomputed-Branch architecture, this improves the effectiveness of the IJB entries. With partitioning, the cost and performance for this architecture would be identical to the design proposed, with the addition of an extra cycle of delay when instructions are fetched into the instruction cache. One benefit

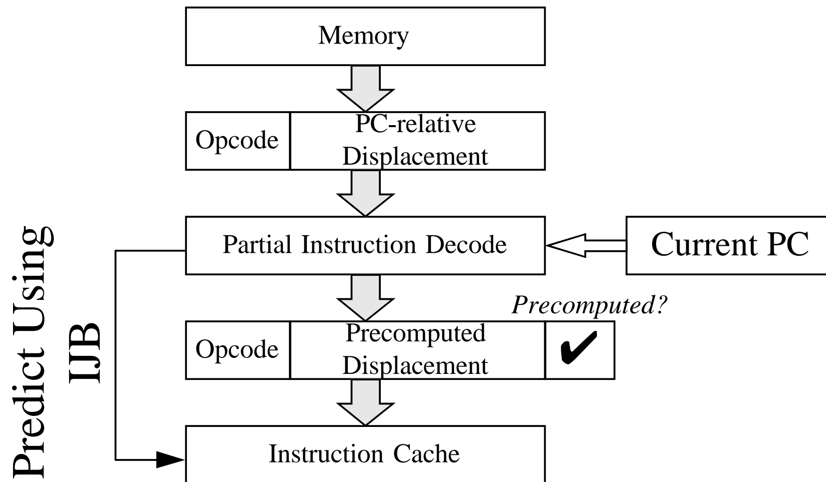


Figure 8: Decoding Instructions When Fetching From Memory

to this alternate design shown in Figure 8, is that the instruction set architecture does not need to be changed since the compiler still creates branches with PC-relative offsets.

8 Conclusions

We have shown that using a pre-computed or non-relative branch displacement is more effective than an architecture that uses a large BTB to cache the destinations for branches. We examined a variety of simple partitioning algorithms that break programs into multiple branch spaces and convert the inter-space branches to indirect jumps. These partitioning algorithms are simple, work well without profile information, and work better with real or estimated profile information. Also, these same partitioning algorithms are already used in existing compilers to map procedures and basic blocks to improve page utilization and to improve instruction cache performance [25, 30]. We have shown that combining a small indirect jump buffer with the Precomputed-Branch architecture results in a branch architecture that uses few resources and has excellent performance, particularly for programs with a large number of branches. We also described how to use the Precomputed-Branch architecture in existing processors without having to modify the instruction set architecture. Lastly, there is another advantage to the proposed designs – the pre-computed branch destination does not depend on the size of the address range. By comparison, the size of a branch target buffer would increase as the instruction address range increases, and will pose problems for 64-bit processors.

This branch encoding has been used in older architectures such as the PDP-8, but is even better suited for modern architectures when a sizable branch displacement field is provided. The Precomputed-Branch architecture exploits the information available at each step of the compilation and execution process. This separates the branch targets that can be pre-computed prior to execution from those that actually need dynamic prediction, such as indirect jumps with multiple targets using an indirect jump buffer and return instructions using the return-stack [19].

Acknowledgments

We would like to thank Alan Eustace and Amitabh Srivastava for providing ATOM, which greatly simplified our work. We would also like to thank Cliff Young for providing helpful comments. This work was funded in part by NSF grant No. ASC-9217394, ARPA contract ARMY DABT63-94-C-0029 and an equipment grant from Digital Equipment Corporation.

References

[1] W. A. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. On the performance enhancement of paging systems through program analysis and transformation. *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[2] Jean Loup Baer and R. Caughey. Segmentation and optimization of programs from Cyclic Structure Analysis. In *Proc. AFIPS*, pages 23–36, 1972.

[3] Thomas Ball and James R. Larus. Branch prediction for free. In *1993 SIGPLAN Confernce on Programming Language Design and Implementation*, pages 300–313. ACM, June 1993.

[4] M. Berry. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[5] Brad Calder and Dirk Grunwald. Fast & accurate instruction fetch and branch prediction. In *21st Annual International Symposium of Computer Architecture*, pages 2–11. ACM, April 1994.

[6] Brad Calder and Dirk Grunwald. Reducing branch costs via branch alignment. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 242–251. ACM, 1994.

[7] Brad Calder and Dirk Grunwald. Reducing indirect function call overhead in C++ programs. In *Proceedings of the 21st Annual ACM Symposium on Principles of Programming Languages*, pages 397–408, January 1994.

[8] Brad Calder and Dirk Grunwald. Next cache line and set prediction. In *22nd Annual International Symposium of Computer Architecture*, pages 287–296. ACM, June 1995.

[9] Brad Calder, Dirk Grunwald, Michael Jones, Donald Lindsay, Jim Martin, Michael Mozer, and Benjamin Zorn. Evidence-based static branch prediction using machine learning. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.

[10] Brad Calder, Dirk Grunwald, and Amitabh Srivastava. The predictability of branches in libraries. In *28th International Symposium on Microarchitecture*, pages 24–34, Ann Arbor, MI, November 1995. IEEE.

[11] Brad Calder, Dirk Grunwald, and Benjamin Zorn. Quantifying behavioral differences between C and C++ programs. *Journal of Programming Languages*, 2(4), 1994.

[12] David R. Ditzel and Hubert R. McLellan. Branch folding in the CRISP microprocessor: Reducing branch delay to zero. In *14th Annual International Symposium of Computer Architecture*, pages 2–9. ACM, ACM, June 1987.

[13] Domenico Ferrari. Improving locality by critical working sets. *Communications of the ACM*, 17(11):614–620, 1974.

[14] J. A. Fisher and S. M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, Boston, Mass., October 1992. ACM.

[15] S. J. Hartley. Compile-time program restructuring in multiprogrammed virtual memory systems. *IEEE Transactions on Software Engineering*, 14(11):1640–1644, 1988.

[16] Amir Hashemi, David R. Kaeli, and Brad Calder. Efficient procedure mapping using cache line coloring. In *Proceedings of the ACM SIGPLAN '97 Conference on Programming Language Design and Implementation*, June 1997.

[17] Mark Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, December 1988.

[18] Peter Yan-Tek Hsu. Designing the TFP microprocessor. *IEEE Micro*, 14(2):23–33, April 1994.

[19] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *18th Annual International Symposium of Computer Architecture*, pages 34–42. ACM, May 1991.

[20] Manolis G. H. Katevenis. *Reduced Instruction Set Computer Architecture for VLSI*. ACM Doctoral Dissertation Award Series. MIT Press, 1985.

[21] Johnny K. F. Lee and Alan Jay Smith. Branch prediction strategies and branch target buffer design. *IEEE Computer*, 21(7):6–22, January 1984.

[22] David J. Lilja. Reducing the branch penalty in pipelined processors. *IEEE Computer*, pages 47–55, July 1988.

[23] Scott McFarling. Combining branch predictors. TN 36, DEC-WRL, June 1993.

[24] Scott McFarling and John Hennessy. Reducing the cost of branches. In *13th Annual International Symposium of Computer Architecture*, pages 396–403. ACM, 1986.

[25] Wen mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *168th Annual International Symposium of Computer Architecture*, pages 242–251. ACM, ACM, 1989.

[26] MIPS Technologies, Incorperated. R10000 microprocessor product overview. Technical report, MIPS Technologies, Incorperated, October 1994.

[27] Johannes M. Mulder, Nhon T. Quach, and Michael J. Flynn. An area model for on-chip memories and its application. *IEEE Journal of Solid-State Circuits*, 26(2):98–105, February 1991.

[28] S.-T. Pan, K. So, and J. T. Rahmeh. Improving the accuracy of dynamic branch prediction using branch correlation. In *Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 76–84, Boston, Mass., October 1992. ACM.

[29] Chris Perleberg and Alan Jay Smith. Branch target buffer design and optimization. *IEEE Transactions on Computers*, 42(4):396–412, April 1993.

[30] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 16–27. ACM, ACM, June 1990.

[31] Steven Przybylski, Mark Horowitz, and John Hennesy. Characteristics of performance-optimal multi-level cache hierarchy. In *168th Annual International Symposium of Computer Architecture*, pages 114–121. IEEE, 1989.

[32] J. E. Smith. A study of branch prediction strategies. In *8th Annual International Symposium of Computer Architecture*, pages 135–148. ACM, 1981.

[33] S. Peter Song, Marvin Denman, and Joe Chang. The PowerPC 604 RISC microprocessor. *IEEE Micro*, 14(5):8–17, October 1994.

[34] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *1994 Programming Language Design and Implementation*, pages 196–205. ACM, June 1994.

[35] Amitabh Srivastava and David W. Wall. A practical system for intermodule code optimizations at link-time. *Journal of Programming Languages*, March 1992. (Also available as DEC-WRL TR-92-6).

[36] Tim A. Wagner, Vance Maverick, Susan Graham, and Michael Harrison. Accurate static estimators for program optimization. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 85–96, Orlando, Florida, June 1994. ACM.

[37] D. W. Wall. Limits of instruction-level parallelism. In *Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 176–188, Boston, Mass., 1991.

[38] Steven J. E. Wilton and Norman P. Jouppi. An enhanced access and cycle time model for on-chip caches. Report 93/5, DEC Western Research Lab, 1993.

[39] Youfeng Wu and James R. Larus. Static branch frequency and program profile analysis. In *27th International Symposium on Microarchitecture*, San Jose, Ca, November 1994. IEEE.

[40] Tse-Yu Yeh and Yale N. Patt. Alternative implementations of two-level adaptive branch predictions. In *19th Annual International Symposium of Computer Architecture*, pages 124–134, Gold Coast, Australia, May 1992. ACM.

[41] Tse-Yu Yeh and Yale N. Patt. A comprehensive instruction fetch mechanism for a processor supporting speculative execution. In *25th International Symposium on Microarchitecture*, pages 129–139, Portland, Or, December 1992. ACM.

[42] Tse-Yu Yeh and Yale N. Patt. A comparison of dynamic branch predictors that use two levels of branch history. In *20th Annual International Symposium of Computer Architecture*, pages 257–266, San Diego, CA, May 1993. ACM.

[43] Cliff Young and Michael D. Smith. Improving the accuracy of static branch prediction using branch correlation. In *6th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 232–241, October 1994.

A Detailed Information From Trace-Driven Simulations

Program	Seperate		Preorder		Prof Depth		Max Cut	
	Seg	%IIB	Seg	%IIB	Seg	%IIB	Seg	%IIB
APS	797	4.63	9/2	3.38	9/1	0.19	8/0	0.19
CSS	818	5.24	9/2	2.40	9/2	2.47	9/0	1.37
LGS	726	10.14	6/0	6.44	6/1	1.36	6/0	1.68
LWS	714	7.03	6/1	3.14	6/1	0.00	6/0	0.00
NAS	740	10.99	7/1	9.47	7/1	0.43	7/0	0.43
OCS	717	0.44	6/1	0.44	6/0	0.06	6/0	0.00
SDS	768	0.37	6/1	0.02	6/0	0.02	6/0	0.14
TFS	715	2.43	6/0	2.30	6/1	0.28	6/0	0.21
TIS	681	0.00	5/1	0.00	5/1	0.00	5/0	0.00
WSS	757	2.09	7/1	0.97	7/0	0.54	7/0	0.67
doduc	708	6.86	6/1	0.21	6/1	1.06	6/0	1.02
fpppp	685	2.66	6/1	0.22	6/3	0.70	6/0	0.00
hydro2d	716	1.38	6/1	0.00	6/1	0.00	6/0	0.00
mdljsp2	733	0.29	6/2	0.27	6/1	0.00	6/0	0.00
nasa7	706	5.95	6/0	0.22	6/2	1.06	6/0	1.08
ora	668	9.75	5/1	0.40	5/2	0.00	5/0	0.00
spice	815	2.28	9/2	1.73	9/2	0.82	9/0	0.33
su2cor	711	6.92	6/0	4.65	7/2	0.02	6/0	0.02
swm256	677	0.08	5/0	0.08	5/1	0.00	5/0	0.00
tomcatv	617	0.03	5/1	0.01	5/1	0.00	5/0	0.00
wave5	762	7.34	7/1	3.19	7/1	7.12	7/0	1.68
alvinn	212	0.64	2/0	0.00	2/0	0.00	2/0	0.00
compress	149	1.95	1/0	0.00	1/0	0.00	1/0	0.00
ear	290	17.42	2/0	0.05	2/0	0.00	2/0	0.00
eqntott	212	0.69	2/0	0.16	2/0	0.00	2/0	0.00
espresso	551	2.26	4/1	1.22	4/0	0.49	4/0	0.11
gcc	1651	4.90	12/2	1.72	12/2	1.67	12/0	1.92
li	551	12.92	3/0	0.84	3/0	0.84	3/0	0.90
sc	512	4.55	4/1	0.75	4/0	0.92	4/0	0.03
cfront	981	7.69	16/13	4.30	15/7	3.70	14/0	4.02
db++	329	6.75	2/0	0.00	2/0	0.00	2/0	0.00
groff	1756	9.00	8/4	3.91	8/2	3.19	8/0	2.03
idl	1459	9.07	5/1	7.72	5/1	7.66	5/0	7.50
lic	5333	12.52	25/5	6.12	24/2	4.44	24/0	3.27
porky	3704	17.74	14/2	13.78	14/2	4.12	14/0	3.30
Overall Avg	940	5.57	7/1	2.29	7/1	1.23	7/0	0.91

Table 7: Efficacy of Program Partitioning with 14-bit Branch Displacements. The format of the table and the significance of the values is discussed in §5.1.

Program	Seperate		Preorder		Prof Depth		Max Cut	
	Seg	%IIB	Seg	%IIB	Seg	%IIB	Seg	%IIB
APS	797	4.63	2/0	2.73	2/0	0.51	2/0	0.00
CSS	818	5.24	3/0	1.76	3/1	2.83	3/0	0.00
LGS	726	10.14	2/0	0.00	2/0	0.00	2/0	0.00
LWS	714	7.03	2/0	0.00	2/0	0.00	2/0	0.00
NAS	740	10.99	2/0	0.00	2/1	0.00	2/0	0.00
OCS	717	0.44	2/0	0.00	2/0	0.00	2/0	0.00
SDS	768	0.37	2/0	0.00	2/0	0.00	2/0	0.00
TFS	715	2.43	2/0	0.00	2/0	0.00	2/0	0.00
TIS	681	0.00	2/1	0.00	2/1	0.00	2/0	0.00
WSS	757	2.09	2/0	0.44	2/0	0.22	2/0	0.00
doduc	708	6.86	2/0	0.00	2/0	0.00	2/0	0.00
fpppp	685	2.66	2/0	0.00	2/0	0.00	2/0	0.00
hydro2d	716	1.38	2/0	0.00	2/0	0.00	2/0	0.00
mdljsp2	733	0.29	2/0	0.00	2/0	0.00	2/0	0.00
nasa7	706	5.95	2/0	0.00	2/0	0.00	2/0	0.00
ora	668	9.75	2/1	0.00	2/1	0.00	2/0	0.00
spice	815	2.28	3/1	1.69	3/0	0.82	3/0	0.00
su2cor	711	6.92	2/0	0.00	2/0	0.00	2/0	0.00
swm256	677	0.08	2/0	0.00	2/0	0.00	2/0	0.00
tomcatv	617	0.03	2/0	0.00	2/0	0.00	2/0	0.00
wave5	762	7.34	2/0	0.02	2/1	2.93	2/0	0.00
alvinn	212	0.64	1/0	0.00	1/0	0.00	1/0	0.00
compress	149	1.95	1/0	0.00	1/0	0.00	1/0	0.00
ear	290	17.42	1/0	0.00	1/0	0.00	1/0	0.00
eqntott	212	0.69	1/0	0.00	1/0	0.00	1/0	0.00
espresso	551	2.26	1/0	0.00	1/0	0.00	1/0	0.00
gcc	1651	4.90	3/0	1.05	3/0	0.89	3/0	0.97
li	551	12.92	1/0	0.00	1/0	0.00	1/0	0.00
sc	512	4.55	1/0	0.00	1/0	0.00	1/0	0.00
cfront	981	7.69	4/0	2.31	4/0	1.99	4/0	1.06
db++	329	6.75	1/0	0.00	1/0	0.00	1/0	0.00
groff	1756	9.00	2/0	1.40	2/1	0.87	2/0	0.84
idl	1459	9.07	2/0	5.04	2/0	5.04	2/0	5.04
lic	5333	12.52	6/0	2.87	6/0	0.37	6/0	0.29
porky	3704	17.74	4/0	2.44	4/0	3.00	4/0	2.75
Overall Avg	940	5.57	2/0	0.62	2/0	0.56	2/0	0.31

Table 8: Efficacy of Program Partitioning with 16-bit Branch Displacements. The format of the table and the significance of the values is discussed in §5.1.

	14 bit Branch Space				16 bit Branch Space				21 bit Branch Space			
Program	No	4	16	32	No	4	16	32	No	4	16	32
APS	6.57	3.89	3.33	3.18	5.91	3.57	3.26	3.14	3.18	3.11	3.09	3.09
CSS	7.62	6.63	6.03	5.50	6.99	6.01	5.29	5.12	5.22	4.55	4.16	4.01
LGS	11.59	5.39	5.17	5.14	5.14	5.14	5.14	5.14	5.14	5.14	5.14	5.14
LWS	8.51	6.30	5.36	5.36	5.36	5.36	5.36	5.36	5.36	5.36	5.36	5.36
NAS	13.62	5.01	3.83	3.32	4.15	3.44	3.31	3.31	4.15	3.44	3.31	3.31
OCS	2.87	2.47	2.45	2.45	2.43	2.42	2.42	2.42	2.43	2.42	2.42	2.42
SDS	3.28	3.28	3.26	3.25	3.26	3.25	3.25	3.25	3.26	3.25	3.25	3.25
TFS	6.32	5.87	3.98	3.91	4.02	3.88	3.86	3.86	4.02	3.87	3.86	3.86
TIS	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66
WSS	6.47	4.53	3.91	3.82	5.93	3.99	3.82	3.82	5.50	3.98	3.82	3.82
doduc	4.60	4.52	4.50	4.47	4.40	4.39	4.39	4.39	4.40	4.39	4.39	4.39
fpppp	4.86	4.63	4.63	4.63	4.63	4.63	4.63	4.63	4.63	4.63	4.63	4.63
hydro2d	2.80	2.80	2.79	2.79	2.79	2.79	2.79	2.79	2.79	2.79	2.79	2.79
mdljsp2	6.96	6.69	6.69	6.69	6.69	6.69	6.69	6.69	6.69	6.69	6.69	6.69
nasa7	2.77	2.40	2.29	2.24	2.55	2.27	2.20	2.20	2.55	2.27	2.20	2.20
ora	3.02	2.61	2.61	2.61	2.61	2.61	2.61	2.61	2.61	2.61	2.61	2.61
spice	5.95	4.56	4.07	4.06	5.91	4.54	4.06	4.06	4.22	4.06	4.06	4.06
su2cor	14.15	9.33	9.32	9.32	9.50	9.32	9.32	9.32	9.50	9.32	9.32	9.32
swm256	0.67	0.52	0.52	0.52	0.59	0.52	0.52	0.52	0.59	0.52	0.52	0.52
tomcatv	0.51	0.51	0.49	0.49	0.50	0.50	0.48	0.48	0.50	0.50	0.48	0.48
wave5	6.76	3.55	3.52	3.48	3.59	3.48	3.48	3.48	3.57	3.48	3.48	3.48
alvinn	0.23	0.21	0.21	0.21	0.23	0.21	0.21	0.21	0.23	0.21	0.21	0.21
compress	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86
ear	4.03	3.94	3.94	3.94	3.98	3.94	3.94	3.94	3.98	3.94	3.94	3.94
eqntott	3.13	1.31	1.28	1.28	2.97	1.31	1.28	1.28	2.97	1.31	1.28	1.28
espresso	6.47	5.54	5.29	5.16	5.25	5.09	5.07	5.07	5.25	5.09	5.07	5.07
gcc	15.14	14.01	13.42	13.06	14.47	13.42	12.97	12.68	13.42	12.48	12.33	12.32
li	5.92	4.80	4.26	4.08	5.08	4.02	4.01	4.01	5.08	4.02	4.01	4.01
sc	5.33	4.15	3.95	3.89	4.42	3.83	3.82	3.82	4.42	3.83	3.82	3.82
cfront	18.69	18.08	17.12	16.42	16.70	16.14	15.74	15.27	14.39	13.93	13.75	13.66
db++	15.71	0.78	0.76	0.76	15.71	0.78	0.76	0.76	15.71	0.78	0.76	0.76
groff	11.42	9.55	7.29	6.59	8.91	7.49	6.14	5.82	7.51	6.34	5.45	5.31
idl	21.07	20.87	3.38	2.59	18.39	18.25	2.99	2.19	13.35	13.21	2.20	1.85
lic	13.02	10.39	9.56	8.98	9.77	8.46	8.00	7.31	6.90	6.77	6.74	6.74
porky	19.55	8.51	6.40	5.42	8.21	6.18	5.09	4.42	5.77	4.18	3.53	3.26
Overall Avg	7.75	5.69	4.78	4.60	6.07	5.13	4.53	4.43	5.45	4.69	4.29	4.26

Table 9: The percentage of mispredicted branches using the Preorder partitioning. The results are shown for a varying number of IJB entries; the IJB entries are only used to predict indirect jumps.

Program	14 bit Branch Space				16 bit Branch Space				21 bit Branch Space			
	No	4	16	32	No	4	16	32	No	4	16	32
APS	3.38	3.18	3.15	3.13	3.18	3.11	3.09	3.09	3.18	3.11	3.09	3.09
CSS	6.60	5.50	4.27	4.08	5.22	4.55	4.16	4.01	5.22	4.55	4.16	4.01
LGS	6.82	5.54	5.14	5.14	5.14	5.14	5.14	5.14	5.14	5.14	5.14	5.14
LWS	5.36	5.36	5.36	5.36	5.36	5.36	5.36	5.36	5.36	5.36	5.36	5.36
NAS	4.58	3.58	3.32	3.32	4.15	3.44	3.31	3.31	4.15	3.44	3.31	3.31
OCS	2.44	2.43	2.42	2.42	2.43	2.42	2.42	2.42	2.43	2.42	2.42	2.42
SDS	3.41	3.27	3.25	3.25	3.26	3.25	3.25	3.25	3.26	3.25	3.25	3.25
TFS	4.24	4.12	3.95	3.88	4.02	3.87	3.86	3.86	4.02	3.87	3.86	3.86
TIS	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66	1.66
WSS	6.17	4.15	3.83	3.82	5.50	3.98	3.82	3.82	5.50	3.98	3.82	3.82
doduc	5.42	4.86	4.39	4.39	4.40	4.39	4.39	4.39	4.40	4.39	4.39	4.39
fpppp	4.63	4.63	4.63	4.63	4.63	4.63	4.63	4.63	4.63	4.63	4.63	4.63
hydro2d	2.79	2.79	2.79	2.79	2.79	2.79	2.79	2.79	2.79	2.79	2.79	2.79
mdljsp2	6.69	6.69	6.69	6.69	6.69	6.69	6.69	6.69	6.69	6.69	6.69	6.69
nasa7	3.63	3.14	2.51	2.24	2.55	2.27	2.20	2.20	2.55	2.27	2.20	2.20
ora	2.61	2.61	2.61	2.61	2.61	2.61	2.61	2.61	2.61	2.61	2.61	2.61
spice	4.55	4.31	4.06	4.06	4.22	4.06	4.06	4.06	4.22	4.06	4.06	4.06
su2cor	9.52	9.33	9.32	9.32	9.50	9.32	9.32	9.32	9.50	9.32	9.32	9.32
swm256	0.59	0.52	0.52	0.52	0.59	0.52	0.52	0.52	0.59	0.52	0.52	0.52
tomcatv	0.50	0.50	0.48	0.48	0.50	0.50	0.48	0.48	0.50	0.50	0.48	0.48
wave5	5.26	3.97	3.48	3.48	3.57	3.48	3.48	3.48	3.57	3.48	3.48	3.48
alvinn	0.23	0.21	0.21	0.21	0.23	0.21	0.21	0.21	0.23	0.21	0.21	0.21
compress	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86	9.86
ear	3.98	3.94	3.94	3.94	3.98	3.94	3.94	3.94	3.98	3.94	3.94	3.94
eqntott	2.97	1.31	1.28	1.28	2.97	1.31	1.28	1.28	2.97	1.31	1.28	1.28
espresso	5.36	5.16	5.10	5.09	5.25	5.09	5.07	5.07	5.25	5.09	5.07	5.07
gcc	15.34	14.08	13.36	12.98	14.38	13.18	12.73	12.53	13.42	12.48	12.33	12.32
li	5.99	4.87	4.27	4.08	5.08	4.02	4.01	4.01	5.08	4.02	4.01	4.01
sc	4.46	3.84	3.83	3.82	4.42	3.83	3.82	3.82	4.42	3.83	3.82	3.82
cfront	18.41	17.48	16.40	15.64	15.45	14.92	14.59	14.38	14.39	13.93	13.75	13.66
db++	15.71	0.78	0.77	0.76	15.71	0.78	0.77	0.76	15.71	0.78	0.76	0.76
groff	9.54	7.98	6.60	6.14	8.36	6.92	5.91	5.69	7.51	6.34	5.45	5.31
idl	20.85	20.60	3.06	2.51	18.39	18.25	2.99	2.19	13.35	13.21	2.20	1.85
lic	10.17	8.71	8.30	7.68	7.19	6.84	6.77	6.76	6.90	6.77	6.74	6.74
porky	9.07	7.16	5.62	4.85	8.52	6.58	5.32	4.65	5.77	4.18	3.53	3.26
Overall Avg	6.37	5.37	4.58	4.46	5.77	4.97	4.42	4.35	5.45	4.69	4.29	4.26

Table 10: The percentage of mispredicted branches using the MaxCut partitioning. The results are shown for a varying number of IJB entries; the IJB entries are only used to predict indirect jumps.