

دانشكده مهندسی کامپيوتر

ارزيابي و تحليل توان مصرفي برنامه هاي مختلف محك استاندارد MiBench بر روي يك سيستم تعبيه شده مبتني بر پردازنده هاي ARM

پايان‌نامه برای دريافت درجه کارشناسي

در رشته مهندسی کامپيوتر - گرايش سخت­افزار

نام دانشجو:

زینب مهدوی

استاد راهنما:

دكتر مهدی فاضلی

آبان ماه 1392



دانشكده مهندسی کامپيوتر

ارزيابي و تحليل توان مصرفي برنامه هاي مختلف محك استاندارد MiBench بر روي يك سيستم تعبيه شده مبتني بر پردازنده هاي ARM

پايان‌نامه برای دريافت درجه کارشناسي

در رشته مهندسی کامپيوتر - گرايش سخت­افزار

نام دانشجو:

زینب مهدوی

استاد راهنما:

دكتر مهدی فاضلی

آبان ماه 1392



تأييديه‌ هيأت داوران جلسه‌ دفاع از پايان‌نامه/رساله

نام دانشكده: مهندسی کامپيوتر

نام دانشجو: زینب مهدوی

عنوان پايان‌نامه: ارزيابی و تحليل توان مصرفی برنامه­های مختلف محک استاندارد MiBench بر روی يک سيستم تعبيه شده مبتني بر پردازنده­های ARM

تاريخ دفاع: آبان ماه 1392

رشته: مهندسی کامپيوتر

گرايش: سخت­افزار

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| رديف | سمت | نام و نام خانوادگي | مرتبه دانشگاهي | دانشگاه يا مؤسسه | امضاء |
| 1 | استاد راهنما | مهدی فاضلی | استاديار | دانشگاه علم و صنعت ايران |  |
| 2 | استاد مدعو داخلي | سید وحید ازهری | استاديار | دانشگاه علم و صنعت ايران |  |

تأييديه‌ صحت و اصالت نتايج

**بسمه تعالی**

اينجانب زینب مهدوی به شماره دانشجويی 88522243 دانشجوی رشته کامپيوتر، گرايش سخت­افزار در مقطع تحصيلی کارشناسی ارشد تأييد می‌نمايم كه كليه‌ی نتايج اين پايان‌نامه حاصل كار اينجانب و بدون هرگونه دخل و تصرف است و موارد نسخه‌برداری‌شده از آثار ديگران را با ذكر كامل مشخصات منبع ذكر كرده‌ام. درصورت اثبات خلاف مندرجات فوق، به تشخيص دانشگاه مطابق با ضوابط و مقررات حاكم (قانون حمايت از حقوق مؤلفان و مصنفان و قانون ترجمه و تكثير كتب و نشريات و آثار صوتي، ضوابط و مقررات آموزشی، پژوهشی و انضباطی) با اينجانب رفتار خواهد شد و حق هرگونه اعتراض در خصوص احقاق حقوق مكتسب و تشخيص و تعيين تخلف و مجازات را از خويش سلب مي‌نمايم. در ضمن، مسئوليت هرگونه پاسخگويی به اشخاص اعم از حقيقي و حقوقي و مراجع ذی‌صلاح (اعم از اداری و قضايی) به عهده اينجانب خواهد بود و دانشگاه هيچ‌گونه مسؤوليتی در اين خصوص نخواهد داشت.

نام و نام خانوادگي: زینب مهدوی

امضا و تاريخ:

مجوز بهره‌برداری از پايان‌نامه

بهره‌برداری از اين پايان‌نامه در چارچوب مقررات كتابخانه و با توجه به محدوديتی كه توسط استاد راهنما به شرح زير تعيين مي‌شود، بلامانع است:

🞎 بهره‌برداری از اين پايان‌نامه براي همگان بلامانع است.

🞎 بهره‌برداری از اين پايان‌نامه با اخذ مجوز از استاد راهنما، بلامانع است.

🞎 بهره‌برداری از اين پايان‌نامه تا تاريخ .................................... ممنوع است.

نام استاد يا اساتيد راهنما:

تاريخ:

امضا:

تقديم به ....(اختياري)

با تشکر از ...(اختياري)

چکيده

يكی از پارامترهای مهم در طراحی سيستم­های تعبيه شده ميزان توان مصرفی در آن­هاست. اين مسئله عمدتا به دو دليل است: 1) اين سيستم­ها برای كار از باطری استفاده مي­كنند، 2) وزن، حجم و هزينه­ی آن­ها بسيار مهم است، لذا نمي­توان از وسايل خنك­كننده­ی مرسوم در سيستم­های ديجيتال در آن­ها استفاده نمود. در اين پروژه هدف بر اين است كه كل برنامه­های محک بسته MiBench كه يک بسته استاندارد برای كاربردهای تعبيه شده­است بر روی يک سيستم تعبيه شده مبتنی بر ARM اجرا شده و ميزان توان مصرفي آن­ها به ازای قسمت­های مختلف برنامه استخراج گرديده و تحليل شود.

واژه‌های كليدی:سیستم­های تعبیه شده، محک، Mibench

فهرست مطالب

|  |  |
| --- | --- |
| عنوان | صفحه |

[فصل 1: مقدمه 1](#_Toc371799720)

[1-1- شرح مسأله 2](#_Toc371799721)

[فصل 2: تعاريف و مفاهيم مبنايي 3](#_Toc371799722)

[2-1- مقدمه 4](#_Toc371799723)

[2-2- توصیف محک 6](#_Toc371799724)

[2-2-1- اتوماسیون و کنترل صنعتی 7](#_Toc371799725)

[2-2-2- شبکه 8](#_Toc371799726)

[2-2-3- امنیت 9](#_Toc371799727)

[2-2-4- دستگاه­های مصرف کننده 10](#_Toc371799728)

[2-2-5- اتوماسیون اداری 12](#_Toc371799729)

[2-2-6- ارتباطات 12](#_Toc371799730)

[2-3- معتبرسازی مدل میکرومعماری 14](#_Toc371799731)

[2-4- تحلیل محک 14](#_Toc371799732)

[2-4-1- توزیع دستورالعمل­ها 16](#_Toc371799733)

[2-4-2- انشعابات 18](#_Toc371799734)

[2-4-3- حافظه 19](#_Toc371799735)

[2-4-4- عملکرد محک 22](#_Toc371799736)

[2-5- نتیجه­گیری 24](#_Toc371799737)

[فصل 3: مروری بر كارهای مرتبط 25](#_Toc371799738)

[3-1- مقدمه 26](#_Toc371799739)

[3-2- آشنایی با شبیه­ساز MEET 26](#_Toc371799740)

[3-2-1- درباره­ی MEET 26](#_Toc371799741)

[3-2-2- بستر سخت­افزاری 27](#_Toc371799742)

[3-2-3- سیستم مورد نیاز 28](#_Toc371799743)

[3-2-4- استفاده از MEET 28](#_Toc371799744)

[3-2-5- گزینه­ها 29](#_Toc371799745)

[3-2-6- مشخصه­های شبیه­سازی 29](#_Toc371799746)

[3-2-7- شرح نتایج شبیه­ساز MEET 30](#_Toc371799747)

[فصل 4: روش/فن/طرح پيشنهادی 31](#_Toc371799748)

[4-1- مقدمه 32](#_Toc371799749)

[4-2- شرح برنامه­ها و نتایج شبیه­سازی آن­ها 32](#_Toc371799750)

[4-2-1- Bitcount 32](#_Toc371799751)

[4-2-2- دیکسترا 34](#_Toc371799752)

[4-2-3- Stringsearch 37](#_Toc371799753)

[4-2-4- بلوفیش 40](#_Toc371799754)

[4-2-5- Rijndael 43](#_Toc371799755)

[4-2-6- Sha 47](#_Toc371799756)

[فصل 5: ارزيابي نتایج شبیه سازی 50](#_Toc371799757)

[5-1- مقدمه 51](#_Toc371799758)

[5-2- ارزیابی نتایج شبیه سازی الگوریتم­های مختلف 51](#_Toc371799759)

[5-2-1- دیکسترا 51](#_Toc371799760)

[5-5-2- Stringsearch 51](#_Toc371799761)

[5-2-3- بلوفیش 52](#_Toc371799762)

[5-2-4- Rijndael 53](#_Toc371799763)

[5-2-5- Sha 53](#_Toc371799764)

[5-2-6- مقایسه­ی الگوریتم­های بلوفیش و Rijndael 54](#_Toc371799765)

[فصل 6: نتيجه‌گيری و کارهای آينده 55](#_Toc371799766)

[6-1- نتيجه‌گيری 56](#_Toc371799767)

[6-2- کارهاي آينده 56](#_Toc371799768)

[مراجع 57](#_Toc371799769)

[واژه نامه 59](#_Toc371799770)

فهرست شکل‌ها

|  |  |
| --- | --- |
| عنوان | صفحه |

[شکل (2-1) توزیع دستورالعمل­های دینامیک برای مجموعه داده­های بزرگ 16](#_Toc371714174)

[شکل (2-2) اندازه­ی ایستای بلوک اولیه در برنامه­های Mibench 18](#_Toc371714175)

[شکل (2-3) نرخ پیش­بینی انشعاب برای محک­های Mibench و SPEC2000 19](#_Toc371714176)

[شکل (2-4) اندازه­های بخش متن و داده و نرخ پیش­بینی برای برخی از محک­ها 20](#_Toc371714177)

[شکل (2-5) نرخ خطای کش برای الگوریتم­های Rijndael و Ispell با خطوط 16 بیتی 21](#_Toc371714178)

[شکل (2-6) نرخ خطای کش برای الگوریتم tiff2rgba با خطوط 16 بیتی 22](#_Toc371714179)

[شکل (2-7) تعداد دستورالعمل­ها در هر چرخه (IPC) 23](#_Toc371714180)

[شکل (3-1) بستر سخت­افزاری شبیه­سازی شده توسط MEET 28](#_Toc371714181)

[شکل (3-2) گزینه­های جدید MEET 29](#_Toc371714182)

[شکل (3-3) مشخصه­های جدید شبیه­سازی 30](#_Toc371714183)

فهرست جدول‌ها

|  |  |
| --- | --- |
| عنوان | صفحه |

[جدول (2-1) محک­های Mibench 6](#_Toc371799393)

[جدول (2-2) سایز دستورالعمل­های محک­های Mibench 15](#_Toc371799394)

[جدول (2-3) تنظیمات ARM 24](#_Toc371799395)

[جدول (4-1) نتایج شبیه­سازی برای الگوریتم Bitcount 34](#_Toc371799396)

[جدول (4-2) نتایج شبیه­سازی الگوریتم Dijkstra، برای ورودی با سایز کوچک 36](#_Toc371799397)

[جدول (4-3) نتایج شبیه­سازی الگوریتم Dijkstra، برای ورودی با سایز بزرگ 36](#_Toc371799398)

[جدول (4-4) نتایج شبیه­سازی الگوریتم Stringsearch، برای ورودی با سایز کوچک 38](#_Toc371799399)

[جدول (4-5) نتایج شبیه­سازی الگوریتم Stringsearch، برای ورودی با سایز بزرگ 39](#_Toc371799400)

[جدول (4-6) نتایج شبیه­سازی الگوریتم بلوفیش، برای ورودی با سایز کوچک 41](#_Toc371799401)

[جدول (4-7) نتایج شبیه­سازی الگوریتم بلوفیش، برای ورودی با سایز بزرگ 42](#_Toc371799402)

[جدول (4-8) نتایج شبیه­سازی الگوریتم Rijndael، برای ورودی با سایز کوچک 45](#_Toc371799403)

[جدول (4-9) نتایج شبیه­سازی الگوریتم Rijndael، برای ورودی با سایز بزرگ 46](#_Toc371799404)

[جدول (4-10) نتایج شبیه­سازی الگوریتم Sha، برای ورودی با سایز کوچک 47](#_Toc371799405)

[جدول (4-11) نتایج شبیه­سازی الگوریتم Sha، برای ورودی با سایز بزرگ 48](#_Toc371799406)

1. مقدمه
   1. شرح مسأله

امروزه سیستم­های تعبیه شده به میزان زیادی در حوزه­های مختلف مورد استفاده قرار می­گیرند، به طوری که بنابر برخی گزارشات، بیش از 99% پردازنده­های تولید شده در سیستم­های تعبیه شده استفاده شده­اند. تعاریف متعددی برای سیستم­های تعبیه شده در ادبیات موضوعی وجود دارد. به طور كلی، منظور از يک سيستم تعبيه شده، سيستمی ديجيتالی است كه كنترل يک وظيفه را در سيستمی بزرگ­تر به عهده می­گيرد و تنها انجام همين يک وظيفه را به عهده دارد. در این حالت گفته می­شود که این سیستم دیجیتال در سیستم میزبان تعبیه شده است. دامنه کاربرد سیستم­های مذکور از کاربردهای خاص، مثل کاربردهای فضایی و هواپیمایی به کاربردهای عمومی­تر در زندگی بشر و همچنین بسیاری از سیستم کاربردهای مالی همانند کاربردهای صنعتی گسترش پیدا کرده است.

يكي از پارامترهای مهم در طراحی سيستم­های تعبيه شده ميزان توان مصرفی در آن­هاست. اين مسئله عمدتا به دو دليل است: 1) اين سيستم­ها برای كار از باطری استفاده مي­كنند، 2) وزن، حجم و هزينه در آن­ها بسيار مهم است، لذا نمی­توان از وسايل خنك­كننده مرسوم در سيستم­های ديجيتال در آن­ها استفاده نمود. در اين پروژه هدف بر اين است كه كل برنامه­های محک بسته MiBench كه يک بسته استاندارد برای كاربردهای تعبيه شده است بر روی يک سيستم تعبيه شده مبتني بر ARM اجرا شده و ميزان توان مصرفی آن­ها به ازای قسمت­های مختلف برنامه استخراج گرديده و تحليل شود. با استفاده از اين تحليل می­توان يک راهنما برای نحوه­ی توسعه برنامه­ی اين سيستم­ها ارائه داد.

1. تعاريف و مفاهيم مبنايي
   1. مقدمه

طراحی مبتنی بر عملکرد، محک­زنی را به یک قسمت بحرانی فرآیند طراحی تبدیل کرده­است. محدوده­ی وسیعی از محک­ها شامل Dhrystone، LINPACK، Whetstone، CPU2، Mediabench و تعداد زیاد دیگری ارائه شده­اند. بیشتر این محک­ها به سمت نواحی خاصی از محاسبات کامپیوتری هدف گرفته شده­اند. برای مثال Dhrystone برای سیستم­های با عملکرد عدد صحیح و یا LINPACK برای محاسبات برداری است. CPU2 و Whetstone برای کاربردهای ممیز شناور[[1]](#footnote-1) عددی و Mediabench برای کاربردهای مالتی مدیا هستند. محک­های دیگری نیز برای تاکید بر TCP/IP شبکه، ورود و خروج دیتا و عملکردهای خاص در دسترس هستند.

بیشترین محک­های مورد استفاده، مربوط به موسسه ارزیابی عملکرد استاندارد[[2]](#footnote-2) CPU یا به عبارتی SPEC است، که در حال حاضر در سومین بازبینی خود هستند (SPEC2000). آن­ها با فراهم آوردن یک مجموعه­ی خود شامل از برنامه­ها و دیتاها به دسته­های ممیز شناور و عدد صحیح جداگانه، یک حجم کاری برای کامپیوترهای همه منظوره[[3]](#footnote-3) مشخص می­کنند.محبوبیت محک­های SPEC به عنوان یک معیار عملکرد به طور موثری تحت تاثیر طراحی میکروپروسسورهای همه منظوره است، به خصوص آن­هایی که در سرورها و سیستم­های رومیزی High-end استفاده می­شوند.از میان مشخصه­های عمومی این این سیستم­ها، پایپ لاین­های عمیق، موازی سازی سطوح دستورالعمل­ها، پیش بینی محک­های پیچیده و حافظه­های کش بزرگ هستند.

این گروه از ماشین­ها در مرکز توجه جامعه مجازی کامپیوتر قرار دارند. تعداد کمی از میکروپروسسورها در این بخش از بازار استفاده شده­اند. حجم زیادی از میکروپروسسورها در کاربردهای تعبیه شده[[4]](#footnote-4) استفاده می­شوند. اگرچه تعداد زیادی از آن­ها فقط میکروکنترلرهای ارزان هستند، فروش آن­ها تقریبا نزدیک به نصف سود تمام میکروپروسسورهاست. علاوه بر این دامنه­ی کاربرد تعبیه شده پرسرعت ترین بخش بازار در صنعت میکروپروسسور است.

محدوده­ی وسیع برنامه­ها، مشخص کردن محدوده­ی سیستم­های تعبیه شده را دشوار کرده­است. در حقیقت یک مجموعه­ی محک تعبیه شده باید این تفاوت­ها را منعکس کند. محدوده­ی کاربردهای آن از سیستم­های سنسور در میکروکنترلرهای ساده تا تلفن­های سلولار هوشمند که عملکرد یک ماشین رومیزی مرکب با پشتیبانی ارتباطات وایرلس را دارد، گسترده است. شاید تنها تقسیم کننده­ها موارد زیر باشند:

1. پروسسورهای تعبیه شده معمولا برای اینکه به طور همزمان در پروسه طراحی به حساب بیایند، نیاز به توان دارند.
2. یک پایگاه کد مهم وجود ندارد که بتواند یک معماری مجموعه دستورالعمل­های استاندارد را مساعدت کند. این اتفاق منجر به افزایش قابل توجه در تعداد ISA ها در کاربردهای تعبیه شده شده و این تعداد در حال رشد است.

تلاش­هایی برای مشخص کردن حجم­های کاری سیستم­های تعبیه شده انجام شده است که مقدار قابل توجهی از آن­ها به وسیله [[5]](#footnote-5)EEMBC تهیه شده است. آن­ها متوجه سختی مشکلات استفاده از تنها یک مجموعه برای مشخصه نگاری چنین کاربردهای متنوعی شده­اند و به جای آن یک مجموعه از مجموعه­هایی که در پنج بازار تعبیه شده وجود دارد، حجم کار را نمونه می­گیرد. متاسفانه محک­های EEMBC به جز محک SPEC، به دلیل هزینه بالای عضو شدن در کنسرسیون، به سهولت برای تحقیقات دانشگاهی قابل دسترسی نیستند.

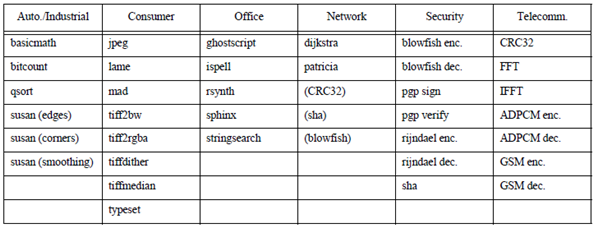
در این مقاله ما یک مجموعه 35 تایی از کاربردهای تعبیه شده را برای اهداف محک زنی که با نام Mibench نام برده می­شوند، ارائه می­کنیم. بر اساس مدل EEMBC این محک­ها به 6 مجموعه تقسیم می­شوند که هر کدام از این مجموعه­ها یک حوزه خاص از بازار تعبیه شده را هدف قرار می­دهند. این شش سرفصل عبارتند از: اتوماسیون و کنترل صنعتی[[6]](#footnote-6)، قطعات مصرف کننده[[7]](#footnote-7)، اتوماسیون اداری[[8]](#footnote-8)، شبکه[[9]](#footnote-9)، امنیت[[10]](#footnote-10) و ارتباطات[[11]](#footnote-11). کد سورس تمام این برنامه­ها موجود است. از آنجایی که از کاربردهای تعبیه شده گذشته به طور مستقیم به زبان اسمبلی نوشته شده­اند، جمع­آوری یک مجموعه قابل حمل از محک­های حوزه تعبیه شده کار سختی است. اگرچه تمایل کنونی در حوزه­ی تعبیه شده نشان می­دهد کامپایلرها در ساده­ترین میکروکنترل­ها و بهترین DSP ها از لحاظ عملکرداستفاده می­شوند. بنابراین Mibench قابل انتقال به هر بستر نرم افزاری[[12]](#footnote-12)­ای که پشتیبانی کامپایلر را دارد، می­باشد.

در ادامه­ِ­ی این بخش می­خوانیم که بخش 2-2 محک­ها و مجموعه داده­ها در Mibench را توصیف می­کند. در بخش 2-3 میکرومعماری مدل ARM، میکرو معماری هسته­ی SA1 معتبرسازی می­شود.این یک مرحله­ی مهم است که معمولا از بحث عملکرد محک در محک­ها حذف می­شود. بخش 2-4 یک تحلیل از محک­های Mibench ارائه می­کند و آن­ها را با محک­ها SPEC2000 مقایسه می­کند. همچنان که هرکس ممکن است از روشی که انتخاب شده است انتظار داشته باشد که برنامه­های Mibench تنوع بیشتری در رفتار در میان مجموعه­ی کل و حوزه­های اختصاصی از خود نشان دهند. همین مسئله نشان می­دهد که SPEC برای راه­اندازی طراحی میکروپروسسورهای آینده برای بسیاری از کاربردهای تعبیه شده موجود در Mibench حجم کاری مناسبی نیست. توزیع دستورالعمل­ها، پیش­بینی دسته­ها و دسترسی به حافظه­ها همگی امتحان شده­اند.در بخش 2-5 خلاصه­ای از مشخصات حجم کاری تعبیه شده موجود در آزمایش­های ما تهیه شده­اند.[1]

* 1. توصیف محک

Mibench شباهت­های زیادی به مجموعه­ی محک EEMBC که در وب سایت خود[2] شرح داده است، دارد. با این وجود Mibench متشکل از کدهای سورس آزاد و در دسترس است. نام همه­ی وب سایت­ها و نویسندگان در هر دسته نگه­داری می­شود، اما تغییرات جزیی ممکن است برای راحتی حمل و نیز توسعه­ی مجموعه­ی داده­ها به کد سورس اضافه شود. در صورت لزوم امکان استفاده از داده­های کوچک و بزرگ وجود دارد. مجموعه داده­های کوچک و سبک وزن برای برنامه­های محک تعبیه شده مناسب است، در حالی که مجموعه داده­های بزرگ برای کاربرد در دنیای واقعی فراهم شده­اند. Mibench متشکل از شش بخش شامل: اتوماسیون و کنترل صنعتی، شبکه، امنیت، دستگاه­های مصرف­کننده، اتوماسیون اداری و ارتباطات می­باشد. این دسته بندی ویژگی­های مختلف برنامه را ارائه می­دهد که محققان در معماری و کامپایلر را قادر می­سازد به بررسی طرح­های خود به طور موثر برای بازار خاص بپردازند.

جدول (2-1) محک­های Mibench



* + 1. اتوماسیون و کنترل صنعتی[[13]](#footnote-13)

بخش اتوماسیون و کنترل صنعتی برای نشان دادن استفاده از پردازنده­های تعبیه شده در سیستم­های کنترل تعبیه شده در نظر گرفته شده است.این پردازنده­ها نیاز به توانایی در انجام عملیات ریاضی پایه، دستکاری بیت، ورود و خروج داده و سازماندهی داده­های ساده دارند. برنامه­های کاربردی نمونه عبارتند از: کنترل­کننده­ی کیسه­ی هوا، مانیتور عملکرد موتور و سیستم­های حسگر. آزمون­های مورد استفاده برای توصیف شرایط، آزمون ریاضی پایه[[14]](#footnote-14)، آزمون شمارش بیتی[[15]](#footnote-15)، یک الگوریتم مرتب سازی[[16]](#footnote-16)، و یک برنامه­ ی تشخیص شکل[[17]](#footnote-17) می­باشد.

* آزمون ریاضی پایه

این آزمون محاسبات ریاضی ساده­ای هستند که اغلب پشتیبانی سخت­افزاری­ای در پردازنده­های تعبیه شده ندارند. به عنوان مثال، حل تابع مکعب، جذر عدد صحیح و تبدیل زاویه از درجه به رادیان، همه محاسبات لازم برای محاسبه ی سرعت در جاده­ها و سایر مقادیر برداری می­باشد. داده­های ورودی، یک مجموعه­ی ثابت از ثابت­[[18]](#footnote-18)ها می­باشد.

* آزمون شمارش بیتی

این الگوریتم، قابلیت دستکاری بیت­ها در یک پردازنده را به وسیله­ی شمارش تعداد بیت­ها در یک آرایه از اعداد صحیح تست می­کند. این فرآیند از طریق پنج روش قابل انجام است: شمارنده­ی بهینه­شده­ی یک بیت در هر حلقه، شمارش بازگشتی 4 بیتی، شمارش غیر بازگشتی 4 بیتی با استفاده از جدول جستجو[[19]](#footnote-19)، شمارش غیر بازگشتی 8 بیتی با استفاده از جدول جستجو و شیفت دادن و شمارش بیت­ها. در اینجا داده­ی ورودی، یک آرایه­ی صحیح از اعداد صحیح با مقادیر ‘0’ و ‘1’ است.

* الگوریتم مرتب سازی Qsort

این آزمون که یک آرایه­ی بزرگ از رشته را به صورت صعودی مرتب می­کند، با عنوان الگوریتم quick\_sort به خوبی شناخته شده است. مرتب سازی اطلاعات برای یک سیستم بسیار مهم است، چرا که به این طریق اولویت­ها می­تواند ایجاد شود، خروجی بهتر می­تواند تفسیر شود، داده­، داده­ها می­تواند سازمان­دهی شود و اینکه به طور کلی زمان اجرای برنامه­ها کاهش می­یابد. در اینجا مجموعه داده­های کوچک یک لیست از کلمات است و مجموعه داده­های بزرگ یک مجموعه­ی سه تایی است که به نقاط داده­ها اشاره می­کند.

* برنامه­ ی تشخیص شکل سوزان[[20]](#footnote-20)

یک بسته ی تشخیص تصویر است. این روش برای تشخیص گوشه­ها و لبه­ها در تصاویر رزونانس مغناطیسی در مغز گسترش یافته است. این روش نوعی از برنامه­های دنیای واقعی است که برای یک چشم­انداز مبتنی بر برنامه­ی تضمین کیفیت به کار گرفته می­شود. سوزان همچنین تصویر را صاف می­کند و تنظیمات آن مانند آستانه[[21]](#footnote-21)، روشنایی و ... را تنظیم می­کند. داده­های ورودی کوچک برای سوزان، یک تصویر سیاه و سفید از یک مستطیل است، در حالی که داده­های ورودی بزرگ برای آن یک تصویر پیچیده است.

* + 1. شبکه[[22]](#footnote-22)

دسته­ی شبکه نشان­دهنده­ی پردازنده­های تعبیه شده در دستگاه­های شبکه، مانند روترها و سوئیچ­ها است. کاری که توسط این پردازنده­ها انجام می­شود شامل محاسبات کوتاهترین مسیر، درخت، جدول جستجو و داده­های ورودی/خروجی است. از الگوریتم­هایی که برای دسته­ی شبکه به کار برده می­شود می­توان به پیدا کردن کوتاه­ترین مسیر در یک گراف و ایجاد و جستجو کردن درخت ساختمان داده­ی پاتریشیا[[23]](#footnote-23) است. بعضی از محک­هایی که در دسته­های امنیت و ارتباطات استفاده می­شود نیز مربوط به دسته­س شبکه می­باشند، مانند: CRC32، SHA و blowfish. با این وجود آن­ها به صورت جداگانه دسته­بندی می­شوند.

* دیکسترا[[24]](#footnote-24)

محک دیکسترا ساختار یک گراف بزرگ را در ماتریس مجاورت نشان می­دهد و سپس کوچکترین مسیر بین هر دو جفت گره را با تکرار برنامه­ی الگوریتم دیکسترا محاسبه می­کند. الگوریتم دیکسترا یک الگوریتم شناخته شده برای یافتن کوتاه­ترین مسیر است و با O() اجرا می­شود.

* پاتریشیا[[25]](#footnote-25)

درخت پاتریشیا یک ساختمان داده است که در جاهایی که درخت کامل با برگ­های پراکنده وجود دارد استفاده می­شود. اغلب درخت پاتریشیا برای الگوریتم­های مسیر یابی[[26]](#footnote-26) مورد استفاده قرار می­گیرد. داده­ی ورودی برای این محک، یک لیست از ترافیک IP از یک سرور بسیار فعال برای یک دوره­ی 2 ساعته است. شماره­های IP شماره­های تبدیل شده هستند.

* + 1. امنیت[[27]](#footnote-27)

اهمیت امنیت داده­ها همانند اینترنت رو به افزایش است و همچنان برای به دست آوردن محبوبیت در فعالیت­های تجارت الکترونیک پیش می­رود. بنابراین، امنیت نیز در Mibench دسته ای را به خود اختصاص داده است. دسته­ی امنیت شامل چندین الگوریتم متداول برای رمزگذاری و رمزگشایی و درهم­سازی[[28]](#footnote-28) داده­ها می­باشد. یکی از این الگوریتم­ها rijndael است که یک استاندارد رمزگذاری جدید[[29]](#footnote-29) (AES) است. سایر الگوریتم­های امنیتی نیز بلوفیش، پی­جی­پی و SHA هستند.

* الگوریتم رمزگذاری/ رمزگشایی بلوفیش

یک نوع رمزنگاری متقارن بلوک با یک کلید (key) با طول متغیر است. این الگوریتم در سال 1992 توسط بروس اشنایدر توسعه داده شد. از آن­جا که طول کلید آن می­تواند بین 32 بیت تا 448 بیت متغیر باشد، برای رمزنگاری­هی داخلی و صادراتی مناسب است. مجموعه داده­های ورودی در این الگوریتم یک فایل متنی ASCII بزرگ و کوچک از یک مقاله­ی آنلاین است.

* الگوریتم امن درهم­سازی[[30]](#footnote-30) sha

یک الگوریتم امن درهم­سازی است که یک پیام خلاصه­ی 160 بیتی را برای یک داده­ی ورودی ایجاد می­کند. این الگوریتم اغلب برای تبادل امن کلیدهای رمزنگاری و برای ایجاد فضاهای دیجیتال ایجاد می­شود. همچنین این الگوریتم در توابع هش MD4 و MD5 که به خوبی شناخته شده هستند، استفاده می­شود. مجموعه داده­های ورودی برای SHA همانند بلوفیش است.

* الگوریتم رمزگذاری/ رمزگشایی Rijndael

این الگوریتم به عنوان موسسه ملی استاندارد و فناوری­های استاندارد رمزنگاری پیشرفته (AES) انتخاب شده­است، که یک الگوریتم رمزنگاری با گزینه­های 128، 192 و 256 بیتی از کلید و بلوک­هاست. مجموعه داده­های ورودی آن همانند بلوفیش است.

* الگوریتم رمزگذاری/ رمزگشایی پی­جی­پی[[31]](#footnote-31)

یک الگوریتم عمومی رمزنگاری کلید است که توسط فیل زیمرمن[[32]](#footnote-32) ارائه شده­ است. این الگوریتم به شما اجازه می­دهد که با افرادی که هرگز ملاقات نکردید، از طریق امضای دیجیتال و رمزنگاری کلید RSA، ارتباط برقرار کنید. داده­ی ورودی برای هر دو تست بزرگ و کوچک، یک فایل متنی کوچک است. این به این دلیل است که پی­جی­پی معمولا فقط برای تبادل ایمن یک کلید برای رمزنگاری بلوک استفاده می­شود و پس از آن دیتا می­تواند با یک نرخ بسیار سریع رمزنگاری یا رمزگشایی شود.

* + 1. دستگاه­های مصرف کننده[[33]](#footnote-33)

محک دستگاه­های مصرف کننده برای نشان دادن بسیاری از دستگاه­های مصرف کننده­ای که در سال­های گذشته محبوبیت آن­ها افزایش یافته است مانند اسکنر، دوربین­های دیجیتال و دستیاران دیجیتال شخصی[[34]](#footnote-34) (PDA)، در نظر گرفته شده است. این دسته در درجه­ی اول بر برنامه­های کاربردی چند رسانه­ای یا الگوریتم­هایی مانند رمزگذاری و رمزگشایی Jpeg، تبدیل فرمت رنگ تصویر، لرزاندن تصویر، کاهش رنگ، کد گذاری و کد گشایی mp3 و حروف چینی HTML تمرکز دارد.همه­ی محک­های تصویر، از تصاویر بزرگ و کوچک به عنوان داده­ی ورودی استفاده می­کنند.

* رمزگذاری/ رمزگشایی JPEG

یک استاندارد پر اتلاف برای فشرده سازی تصویر است. این استاندارد به این دلیل در mibench قرار دارد که یک الگوریتم برای فشرده سازی و رفع فشرده سازی ارائه می­دهد و معمولا برای دیدن تصاویر تعبیه شده در اسناد استفاده می­شود. داده­ی ورودی برای آن یک تصویر کوچک و بزرگ رنگی است.

* Tiffzbw

یک تصویر رنگی به فرمت TIFF را به یک تصویر سیاه و سفید تبدیل می­کند.

* Tiffzrgba

یک تصویر رنگی با فرمت TIFF را به تصویر TIFF با فرمت رنگ RGA تبدیل می­کند.

* Tiffdither

بیت­مپ­های یک تصویر سیاه و سفید با فرمت tiff را برای کاهش دادن وضوح و سایز تصویر، ترکیب می­کند، با این بها که شفافیت تصویر کاهش می­یابد.

* Tiffmedian

تعداد رنگ­های یک عکس را با چندین بار میانگین گرفتن از رنگ فعلی پالت، کاهش می­دهد.

* Lame

یک رمزگذار MP3 به GPL است که رمزگذاری با نرخ بیت ثابت، متوسط و متغیر را پشتیبانی می­کند. این رمزگذار از فایل­های کوچک و بزرگ موجی برای داده­ی ورودی خود استفاده می­کند.

* Mad

یک رمزگشای صوتی MPEG با کیفیت بالا است که در حال حاضر از فرمت MPEG.1 و MPEG.2 برای نمونه برداری فرکانس­های پایین پشتیبانی می­کند.هر سه لایه صوتی (لایه 1، لایه 2 و لایه 3 که به عنوان mp3 شناخته می­شود)، به طور کامل اجرا شده است. این رمزگشا از mp3 های بزرگ و کوچک برای داده­ی ورودی و استفاده می­کند.

* Typeset

یک ابزار حروف­چینی عمومی است که دارای یک پردازنده­ی front-end برای HTML است. این محک، پردازش مورد نیاز برای یک سند HTML را بدون هیچ گونه سرباری[[35]](#footnote-35) انجام می­دهد. این محک نماینده­ی یک جزء اصلی از یک مرورگر وب است که ممکن است در یک دستگاه مصرف کننده استفاده شود. در اینجا ورودی­های بزرگ و کوچک آگهی انتشار جی­سی­سی و صفحه­ی اصلی وب سیمپل­اسکالر[[36]](#footnote-36) است.

* + 1. اتوماسیون اداری[[37]](#footnote-37)

برنامه­های اداری در درجه­ی اول الگوریتم­هایی برای نشان دادن تشکیلات اداری می­باشد. مانند پرینترها، ماشین­های فکس و واژه پردازها، دستیاران دیجیتال شخصی که در قسمت دستگاه­های مصرف کننده به آن­ها اشاره شد نیز، به شدت به دستکاری داده­ها برای سازمان دهی داده­ها وابسته است.

* Ghostscript

یک مترجم زبان پست­اسکریپ بدون رابط گرافیکی آن است. این محک برای نشان دادن اهمیت رو به رشد قابلیت پست­اسکریپ دستگاه­های تعبیه شده مانند پرینتر ارائه شده است.

* الگوریتم جستجوی رشته[[38]](#footnote-38)

این محک با استفاده از الگوریتم مقایسه­ی حساس به حروف[[39]](#footnote-39) در عبارات به دنبال کلمات داده شده می­گردد.

* Ispell

یک جستجوگر سریع املاست که به جستجوگر یونیکس شباهت دارد، با این تفاوت که سریعتر است. ورودی ispell شامل سندهای بزرگ و کوچک از صفحات وب است.

* Rsynth

یک برنامه سنتز متن به گفتار است که چند تکه کد دامنه عمومی را به صورت یک برنامه ادغام می­کند. ورودی­های بزرگ و کوچک برای آن گزیده­ای از یک مقاله­ی خبری آنلاین هستند.

* Sphinx

یک رمزگشای گفتار است که بر روی قطعات محدود بیان یا گفتار (یک گفتار در واحد زمان) عملیات انجام می­دهد. گفته می­تواند تا چند ده ثانیه طول بکشد. ورودی­های کوچک و بزرگ می­تواند یک دستور ساده و یک دنباله­ی طولانی گفتار باشد.

* + 1. ارتباطات[[40]](#footnote-40)

در کنار دسته­ی دستگاه­های مصرف کننده، به دلیل اهمیت پردازنده­های تعبیه شده­ی مدرن، دسته بندی ارتباطات نیز وجود دارد. با رشد انفجاری اینترنت، بسیاری از دستگاه­های مصرف کننده­ی قابل حمل در حال یکپارچه کردن ارتباطات بی سیم هستند. محک ارتباطات جهت تاکید بر این مسئله در یک دسته بندی جداگانه قرار گرفته است. این محک شامل الگوریتم­های کدگذاری، کدگشایی، تجزیه و تحلیل فرکانس و الگوریتم­های کنترلی[[41]](#footnote-41) می­باشد.

* تبدیل سریع فوریه[[42]](#footnote-42)/ معکوس تبدیل سریع فوریه[[43]](#footnote-43)

این محک بر روی یک آرایه از داده، تبدیل فوریه سریع و تبدیل معکوس را انجام می­دهد. تبدیل فوریه در پردازش سیگنال دیجیتال به منظور پیدا کردن فرکانس­های موجود در سیگنال ورودی داده شده استفاده می­شود. داده­ی ورودی در اینجا یک تابع چند جمله­ای با دامنه­ی شبه تصادفی و اجرای فرکانس سینوسی است.

* رمزگذاری/ رمزگشایی جی­اس­ام[[44]](#footnote-44)

استاندارد جهانی برای ارتباطات موبایل یا جی­اس­ام یک استاندارد برای رمزگذاری / رمزگشایی صدا در اروپا و بسیاری از کشورها می­باشد. این استاندارد از ترکیب TDMA[[45]](#footnote-45) یا FDMA[[46]](#footnote-46) برای رمزگذاری یا رمزگشایی جریان داده­ها استفاده می­کند. داده ورودی در اینجا نمونه­های بزرگ و کوچک گفتار است.

* رمزگذاری/ رمزگشایی مدولاسیون کد پالس دیفرانسیل تطابقی[[47]](#footnote-47)

مدولاسیون کد پالس دیفرانسیل تطابقی یک نوع از استاندارد شناخته شده­ی مدولاسیون کد پالس[[48]](#footnote-48) است. یک اجرای متداول از این محک این است که یک نمونه مدولاسیون کد پالس خطی 16 بیتی را می­گیرد و به یک نمونه 4 بیتی تبدیل می­کند. نرخ فشرده سازی در اینجا 4 به 1 است. داده­های ورودی در اینجا نمونه بزرگ و کوچک از گفتار است.

* کد افزونگی دوره ای 32 بیتی[[49]](#footnote-49) (CRC32)

این محک یک عملیات 32 بیتی کد افزونگی دوره­ای را بر روی یک فایل انجام می­دهد. CRC معمولا برای تشخیص خطا در انتقال داده­ها استفاده می­شود. داده­ی ورودی هم برای آن فایل­های صوتی از محک ADPCM می­باشد.

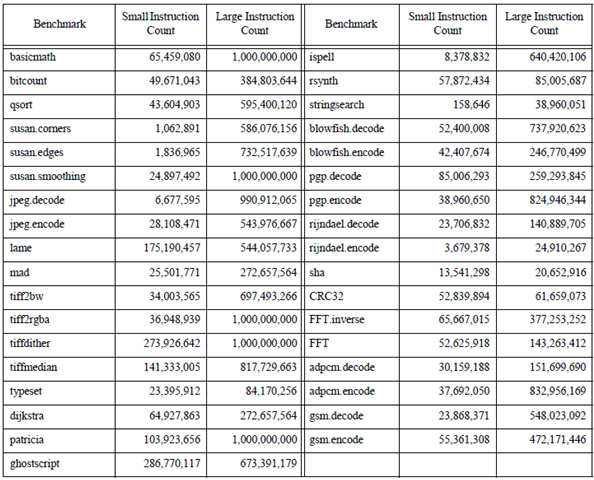
* 1. معتبرسازی مدل میکرومعماری

پیکربندی کنونی در جدول 3 ، از خط لوله­ی strong ARM (SA-1) شرکت اینتل که در سری SA-11XX از ریزپردازنده­های تعبیه شده­ موجود است، مدل شده است. اینتل اطلاعات کمی از خط لوله SA-1 را منتشر کرده است. مدل ما با استفاده از ویژگی­های زمان بندی خط لوله داده شده در راهنمای SA-110 نویسندگان کامپایلر ساخته شده است. به علاوه، ما از ریز محک­ها برای اندازه گیری دقیق تمام تاخیرهای ناشی از خط لوله مانند پیش بینی اشتباه انشعابات و خطای کَش­ها استفاده می­کنیم. ما مدل خود را در برابر ایستگاه کاری توسعه دهنده­ی ریل نت­ویندر[[50]](#footnote-50) معتبر سازی کرده­ایم. نت­ویندر شامل یک ریزپردازنده strongARM SA-110 با فرکانس 275 مگاهرتز، 128 مگابایت حافظه­ی دینامیک[[51]](#footnote-51) و یک رابط اترنت است. نت­ویندر سیستم عامل لینوکس را با یک حلقه ابزار GNU استاندارد که شامل GCC است را اجرا می­کند. زمان اجرای میکرو محک­[[52]](#footnote-52)های صحیح، کرنل­ها (برای مثال FFT) و محک­های بزرگ (مثل BZIP و GCC) در نت­ویندر اندازه گیری شدند و با عملکرد شبیه سازی شده شان در مدل SA-1ARM مقایسه شدند. سادگی خط لوله SA-1 و سیستم حافظه به ما اجازه می­دهد که یک مدل زمانی کاملا دقیق را تنها با مقدار کمی تغییر نسبت به simplescalar/ARM بسازیم. بزرگترین خطای اندازه گیری شده در عملکرد (یا CPI) تنها 3.2 درصد بود. ما نمی­توانستیم به طور کامل مدل کمک پردازنده­ی ممیز شناورمان را معتبرسازی کنیم، زیرا نت­ویندر در سخت افزار از ممیز شناور پشتیبانی نمی­کند. وقتی که بستر های نرم افزاری مرجع و محک­های ممیز شناورمناسب در دسترس باشند، ما این تلاش برای معتبر سازی را بیان خواهیم کرد.

* 1. تحلیل محک

همه­ی محک­ها در SPEC2000 و Mibench به وسیله­ی GCC ورژن 2.95.2 بر روی یک Debian Linux ورژن 2.2.18 همراه با بهینه­سازی­ها کامپایل شده­اند. همه­ی محک­ها به وسیله­ی شبیه­ساز عملکرد SimpleScalar/ARM، با مشخصاتی شبیه به میکروکنترلر Intel Xscale، شبیه­سازی شده­اند. فقط محک­های SPEC2000 صحیح برای مقایسه استفاده شده­اند، زیرا بیشتر پردازنده­های تعبیه شده توانایی­های ممیز شناور مهم را ندارند. یک مجموعه­ی محدود از محک­های SPEC صحیح به طور صحیح روی ARM اجرا شده­اند. بنابراین از این­ها به عنوان نقاط داده استفاده شد، تا یک بیلیون دستورالعمل دینامیک برای همه­ی محک­ها شبیه­سازی شوند. مجموعه­ی داده­ی مرجع برای ورودی SPEC استفاده شد. همچنان که در جدول 2-2 نمایش داده شده، مجموعه­ی داده­های کوچک برای Mibench تقریبا 50 میلیون دستورالعمل دینامیک و برای مجموعه داده­های بزرگ بیش از 750 میلیون دستورالعمل دینامیک است. داده­های عملکرد کش به وسیله­ی شبیه­سازی مراجع حافظه­ی همه­ی محک­ها، با استفاده از چیتا[[53]](#footnote-53) جمع آوری شده­اند. چیتا می­تواند ترکیبات کش چندگانه را در یک مسیر یک­طرفه شبیه سازی کند. پیش­بینی انشعاب با استفاده از sim-bpred شبیه­سازی شده است.

جدول (2-2) سایز دستورالعمل­های محک­های Mibench

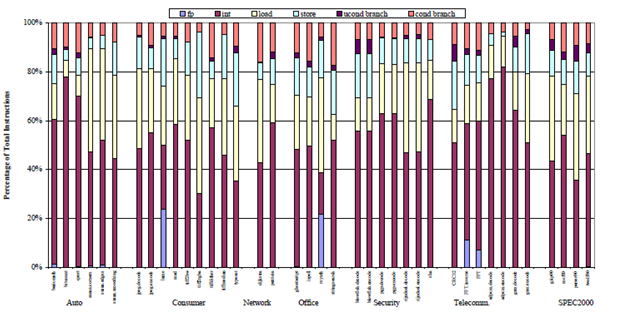


* + 1. توزیع دستورالعمل­ها[[54]](#footnote-54)

چهار کلاس اصلی از دستورالعمل­ها وجود دارد:

* کنترل[[55]](#footnote-55) (انشعابات شرطی و غیر شرطی)
* عدد صحیح[[56]](#footnote-56)
* ممیز شناور[[57]](#footnote-57)
* حافظه[[58]](#footnote-58) (load & store)

در برنامه­های تعبیه شده، محاسبات فشرده، کنترل فشرده، و برنامه­های ورودی/خروجی فشرده وجود دارد. برنامه­های کنترل فشرده درصد بیشتری از دستوراتشان دستورات انشعاب است. در برنامه­های محاسبات فشرده، درصد بیشتر متعلق به دستورات ممیز شناور و عدد صحیح ALU است. برنامه­های IO به اینکه داده­ها در طول انتقال چگونه دستکاری می­شوندبستگی دارد. شکل 2-1 نحوه­ی توزیع تمام برنامه­های Mibench و SPEC2000 را نشان می­دهد.



توزیع دستورالعمل­های دینامیک برای مجموعه داده­های بزرگ

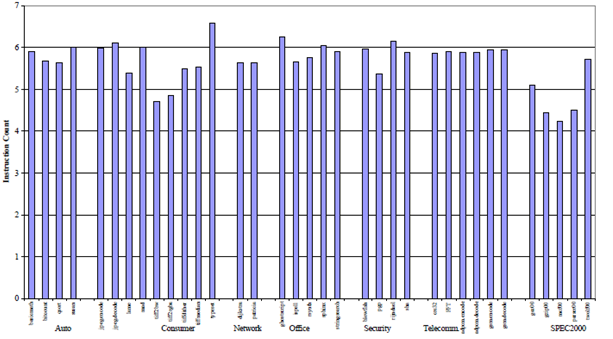
با توجه به شکل، دسته­های محک برخی از این ویژگی­های متمایز را نشان می­دهد. محک­های ارتباطات و امنیت بیشتر از 50 درصد دستورات عدد صحیح ALU را دربر می­گیرند. این برنامه­ها برای پیدا کردن یا تولید آنتروپی در یک مجموعه از داده­ها و به وسیله­ی تکرار عملیات بر روی داده­ها توسعه داده می­شود. محکی مانند محک رمزنگاری/رمزگشاییADPCM در مقایسه با هرکدام از محک­های SPEC که ماکزیمم 50 درصد از دستورات عدد صحیح ALU را دارند، حدوداً 80 درصد از این دستورات را دارند. دسته­ی مصرف­کننده نسبتاً میزان کمی از دستورات عدد صحیح را دارد، ولی بسیاری از دستورات حافظه را انجام می­دهد. این به این دلیل است که داده­های تصویری بزرگ باید پردازش شود. عملیات بر روی هر بخش از تصویر نسبتاً ساده است و دستورات کنترلی کمی نیاز دارد. محک اتوماسیون اداری تعداد زیادی دستورات کنترل و حافظه دارد. این برنامه­ها از فراخوانی توابع کتابخانه­های رشته­ای برای دستکاری داده­های ASCII استفاده می­کند. به این دلیل که داده­ها از نوع متن است، حجم بسیار کمی از حافظه را اشغال می­کند و عملیات حافظه برای ارجاع آن­ها نیاز است. محک­های SPEC تقریباً توزیع یکسانی برای تمام محک­ها دارند.

همان­طور که قبلاً نشان داده شده است، طبقه بندی­های Mibench نشان­دهنده­ی برنامه­های تعبیه شده­ی مختلف هستند. بررسی روی مجموعه­ی کل محک­ها نشان می­دهد که تنوع، وقتی که آن­ها را به صورت کلی در نظر می­گیریم بیشتر است. به عنوان مثال، تعداد انشعابات در Mibench تنوع بسیار کمی دارد. آزمون شمارش بیتی (bitcount)، رمزگذار ADPCM و همچنین چندین محک دیگر، کمتر از 10 درصد از عملیات انشعابی برای برنامه­های محاسبات فشرده استفاده می­کنند. محک­هایی مانند بلوفیش، tiff2rgba و tiffmedian، کمتر از 6 درصد دستورات انشعاب دارند. بیشترین انشعابات متعلق به محک­های متنی است. Stringsearch، ispell و CRC32 در محک ارتباطات در رنجی بین 18 تا 20 درصد از دستورات انشعاب استفاده می­کنند. محک­های SPEC، به جز gzip00 که فقط 9 درصد دستورات انشعاب دارد، معمولاً بیشتر از 15 درصد از دستورات انشعاب استفاده می­کنند. Mibench همچنین در استفاده از عملیات­های حافظه دارای تنوع بیشتری است. بعضی از محک­های آن مانند GSM، tiff2rgba و typeset، بیشتر از 50 درصد از دستورات حافظه را شامل می­شوند، این در حالی است که بقیه، مانند bitcount، رمزگذار ADPCM کمتر از این دستورات استفاده می­کنند. بیشتر محک­های SPEC در حدود 40 درصد از دستورات حافظه استفاده می­کنند.

همچنین نمودار توزیع نشان می­دهد که Mibench، تعداد کمی از دستورات ممیز شناور را در محک­های lame، rsynth و FFT دارد. این­ها تاکیدی بر روی عملیات ممیز شناور ندارند، اما معمولاً نشان داده شده است که در شرایطی، محاسبات ممیز شناور برای کنترل سرعت در جاده­ها، بردار جهت و یا اطلاعات دیگری که برای تعیین عملیات کنترل مورد نیاز است، استفاده می­شود. پردازنده­های فشرده­ی عددی و DSP نیز باید از محک­های ممیز شناور برای جزئیات تجزیه و تحلیل کارایی استفاده کنند.

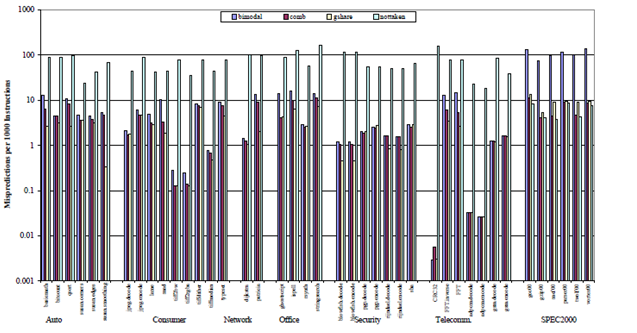
* + 1. انشعابات[[59]](#footnote-59)

Mibench در تعداد انشعابات کاملاً متنوع است. تعداد انشعابات در برخی از محک­ها کم است، به شکل 2-2 توجه کنید. این شکل نشان می­دهد که اندازه­ی ایستای بلوک اولیه در برنامه­های Mibench حدوداً یک دستورالعمل بیشتر از SPEC است. سایز اولیه­ی بلوک در SPECها ، به استثنای twolf00 که بیشتر از 5.5 است، معمولا در حدود 4.5 است. این در حالی است که Mibench تعدادی برنامه با سایز بالای 6 دارد و تقریباً بقیه بالای 5.5 هستند. تعداد کمی از محک­های مصرف­کننده هم هستند که همانند SPECها سایز زیر 5 دارند.



اندازه­ی ایستای بلوک اولیه در برنامه­های Mibench

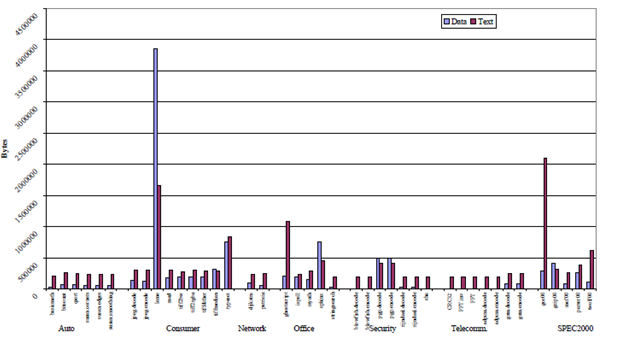
حال که دیدیم Mibench تنوع بیشتری در فرکانس انشعابات نسبت به SPEC2000 دارد، می­توانیم تعیین کنیم که چگونه می­توان به خوبی این انشعابات را پیش­بینی کرد. شبیه­سازی­ها با استفاده از یک طرح بدون پیش­بینی (not-taken)، یک پیش­بینی کننده­ی 8k gshare، یک پیش­بینی کننده­ی 8k bimodal و یک پیش­بینی کننده­ی ترکیبی 8k bimodal/2-level اجرا شد. نرخ پیش­بینی جهت در شکل 2-3 نشان داده شده است. همه­ی پیش­بینی کننده­ها با استثنای استراتژی not-taken از یک 8k BTB[[60]](#footnote-60) استفاده کرده­اند. هیچ افزایش محسوسی در میزان خطای پیش­بینی به دلیل استفاده از BTB رخ نمی­دهد، بنابراین اطلاعاتی از آن در شکل نشان داده نشده است.



نرخ پیش­بینی انشعاب برای محک­های Mibench و SPEC2000

* + 1. حافظه

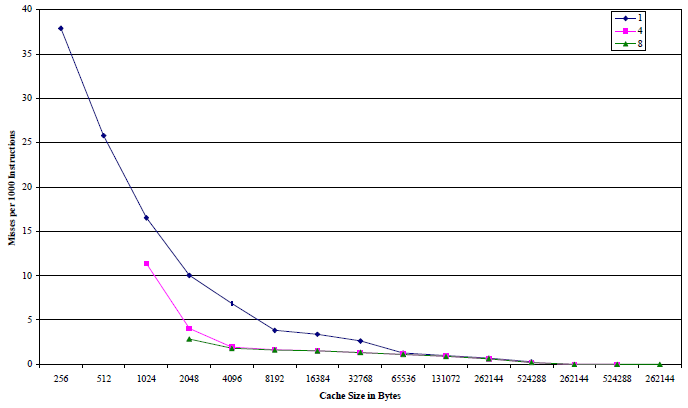
در کنار توزیع دستورالعمل­ها و قابلیت پیش­بینی انشعابات، رفتار حافظه یکی دیگر از نکات مهم قابل ملاحظه به هنگام ارزیابی حجم کار تعبیه شده است. سایز ایستای حافظه و قابلیت کش کردن حافظه با SPEC مقایسه شده است. در شکل 2-4، سایز متن­ها و داده­های Mibench و SPEC2000 نشان داده شده است. این دو مجموعه محک سایز تقریباً مشابهی دارند، اما در اکثر موارد، SPEC2000 بخش­های کمی بزرگ­تری دارد. با این وجود Mibench دارای تنوع بیشتری است. Mibench همانند SPEC2000 که یک محک(gcc00) با بخش متنی­ای با اندازه­ی بزرگ­تر از 2 Mb دارد، یک محک (Ghostscript) با بخش متنی یا اندازه­ای بزرگتر از 1Mb دارد. همچنین Mibench چندین محک با بخش داده­هایی با اندازه­های چندین مگابایتی دارد. بزرگ­ترین داده ی SPEC در محک­هایSPEC2000 اندازه­ای تقریباً برابر با 0.5 Mbدارد (gzip00). این که چرا اندازه­ی بخش داده در محک lame اینقدر زیاد است مشخص نیست، اما ممکن است به این دلیل باشد که به جای محاسبات مجدد در طی فشرده سازی، جداول بزرگ ذخیره می­شوند.

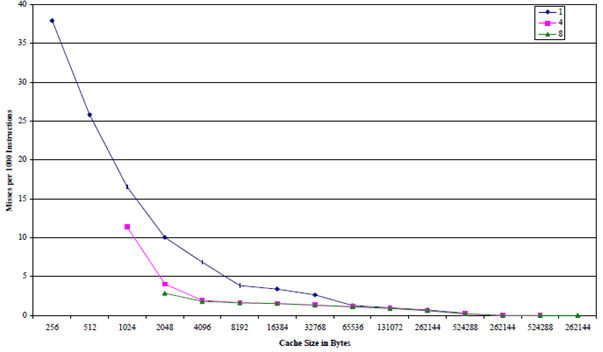


اندازه­های بخش متن و داده و نرخ پیش­بینی برای برخی از محک­ها

تغییرات زیاد در بخش داده­ی Mibench ناشی از تعداد زیادی از مقادیر بلافصل (ثابت) است که در کد سورس تعبیه شده است. هرچند به طور کلی، برنامه­های تعبیه شده حافظه­ی کمی برای بخش­های داده و دستورالعمل دارد که می­تواند اندازه­ی بخش داده و متن به صورت مشترک دیده شود. سایز بخش متنی در حدود 175 تا 200 کیلوبایت بسیار نرمال و اغلب به دلیل گنجاندن کتابخانه­ی استاندارد C است. سایز بخش داده­ها هم به طور کلی حتی کمتر از چند کیلوبایت است.

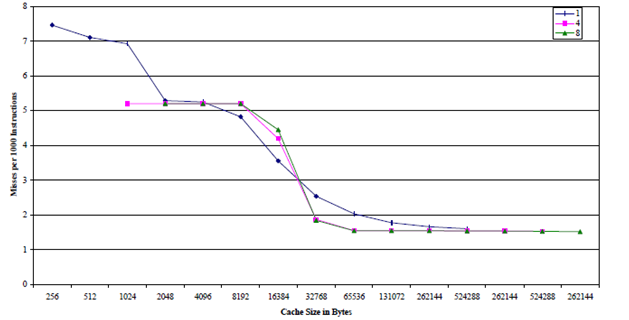
همان­طور که در بالا نشان داده شده است، محک­ها در Mibench سایزهای متفاوتی برای مجموعه­ی داده­ها دارند. به این دلیل، Mibench بر روی برخی از محک­ها نرخ خطای کش مشابه و برای بقیه­ی آن­ها نرخ کم­تری دارد.شکل 2-5 و 2-6، نرخ خطای کش را برای تعداد مجموعه­های متفاوت برای برخی از محک­ها در Mibench نشان داده است. همان­طور که قبلاً اشاره شد، این مقادیر، ماکزیمم نرخ خطا برای هر برنامه­ای است که در مجموعه­های محک شبیه­سازی می­شود. سایر محک­ها نرخ خطای کمتری دارند و بیشتر شبیه شکل 2-5 می­باشند. بیشتر شکل­های نرخ خطا ( برای محک­های مختلف)، با ازای شرکت­پذیری بیشتر از 4-way و سایز بزرگ­تر از 8 کیلوبایت، نرخ خطای ناچیزی دارند.





نرخ خطای کش برای الگوریتم­های Rijndael (شکل بالا) و Ispell (شکل پایین) با خطوط 16 بیتی

همچنین با توجه به شکل 2-5، برای اغلب محک­های Mibench، نرخ خطا تا حدود کمتر از 2 درصد در محدوده­ی 4 تا 6 کیلوبایت به طور زیادی افت می­کنند. برخی از محک­های SPEC2000 نرخ­های خطا را تا محدوده­ی 6 تا 32 کیلوبایت، زیر 2 درصد کاهش نمی­دهد که این مقدار نسبتاً بزرگی است. بعضی از محک­ها مثل gcc00 و patricia به واسطه­ی الگوهای دستیابی تصادفی[[61]](#footnote-61)، نیاز به شرکت­پذیری بیشتری دارند. اما در واقع 8-way برای پایین آوردن نرخ خطا کافی است.



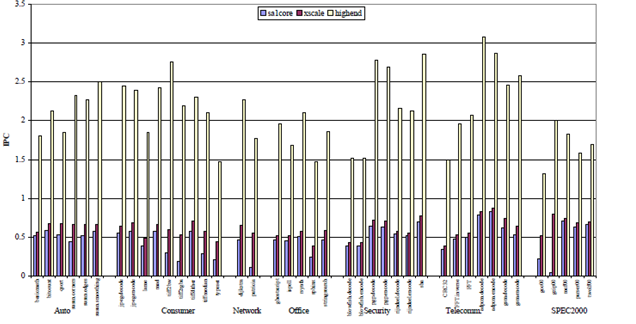
نرخ خطای کش برای الگوریتم tiff2rgba با خطوط 16 بیتی

حافظه­های کش پردازنده­های تعبیه شده ، به جز در کاربردهای مالتی­مدیا، معمولاً کوچک هستند. در برنامه­های تعبیه شده داده­ها مجدداً استفاده می­شوند تا عملکرد کش خوب باشد. مجموعه­ی داده­ها نیز معمولاً ثابت و یا stream-based هستند. کش­های 32-way استفاده شده در ساختار نسل فعلی و نسل آینده برای محک­های شبیه­سازی شده ضروری نیستند. همه­ی محک­های Mibench، نرخ خطاهای کمی با کش­های 4-way یا 8way دارند و تعداد 256 یا 512 برای مجموعه­ها، همان­طور که قبلاً توضیح داده شد کافی است.

* + 1. عملکرد محک

برای انجام یک تحلیل IPC[[62]](#footnote-62) در Mibench، 3 میکروساختار مختلف با simplescalar/ARM شبیه­سازی شده­اند. ساختار این ماشین­ها در جدول 2-3 نمایش داده شده است. ساختار کنونی بعد از انتشار اطلاعات میکرو ساختار ARM SA1 اینتل، مدل شده است. به طور مشابه، ساختار نسل بعدی، بعد از انتشار اطلاعات نسل بعدی میکرو ساختار ARM Xscale مدل شده است. و ساختار High-end نیز بعد از Compaq Alpha 21264 مدل شده است.

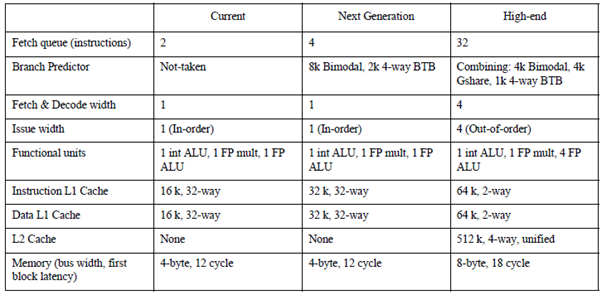
نمایش شبیه­سازی­ها با هر یک از این ساختارها، در شکل 2-7 نمایش داده شده است. بزرگ­ترین مقادیر IPC، به ساخت تصاویر و کاربردهای مرتبط با مالتی­مدیا، مثل tiff2rgba، JPEG decode، tiffmedian، gzip00 و mcf00 برمی­گردد. کمترین مقادیر IPC نیز به blowfish، typeset و CRC32 برمی­گردد که به دلیل طبیعت الگریتم­های رمزگذاری، رمزگشایی و درهم­سازی، وابستگی اطلاعات زیادی دارند. ADPCM و sphinx اگرچه باید وابستگی­های مشابهی داشته باشند، به خوبی مرتبط عمل می­کنند. ساختار high-end به طور محسوسی بهتر از ساختارهای تعبیه شده نسل آینده یا کنونی عمل می­کند. این ساختار به طور نرمال به 2 تا 3 برابر IPC ساختارهای نسل کنونی و آینده دست می­یابد.



تعداد دستورالعمل ها در هر چرخه (IPC)

ساختارهای نسل آینده و کنونی در بیشتر محک­ها عملکرد مشابهی دارند. ساختار نسل آینده خط لوله­ی عمیق­تر، یک پیش­بینی کننده­ی bimodal و دو برابر حافظه­ی کش ساختار کنونی را دارند. از آن­جایی که اغلب انشعابات در Mibench و SPEC به سادگی قابل پیش­بینی هستند، ساختار کنونی از فقدان ساختار موازی­سازی[[63]](#footnote-63) به وسیله­ی طرح not-taken رنج می­برد.. این مسئله به طور نامحسوسی در شکل 2-7 نشان داده شده است، اما به عنوان عملکرد ضعیف سیستم نسل آینده به حساب نمی­اید. همان­طور که قبلاً نشان داده شد، اغلب محک­ها به سادگی cachable هستند و بنابراین ضعف عملکرد نمی­تواند به خاطر مشکلات کش باشد. این ضعف عملکرد باید به خاطر اجرای ترتیبی و نیز فقدان واحدهای عملیاتی باشد. از آن­جا که اغلب محک­ها در Mibench، بلوک­های بایه­ی بزرگ و انشعابات قابل پیش­بینی ساده­ای دارند، احتمالاً منابع کافی برای انجام همه­ی دستورالعمل­های موازی وجود ندارد.

جدول (2-3) تنظیمات ARM



* 1. نتیجه­گیری

طراحی پردازنده­های تعبیه شده برای توسعه­ی یک میکرو ساختار کارامد، نیاز به آگاهی از وظایف تعبیه شده دارد. Mibench در هنگام تحلیل استاتیک و دینامیک خصوصیات عملکرد پردازنده­های تعبیه شده، خصوصیات متفاوت قابل توجهی نسبت به محک­های SPEC2000 دارد. نقشه­ی دستورالعمل دینامیک تغییر بیشتری در تعداد انشعاب­ها، حافظه و عملگرهای عدد صحیح ALU دارد. همچنین سایزهای بخش داده و متن متنوعی دارد. اما داده ها تمایل دارند که بیشتر cachable باشند. Mibench و SPEC2000 هر دو انشعابات قابل پیش­بینی دارند. تنوع تعداد دستورالعمل­ها در هر سیکل نشان می­دهد که محک­ها در دسته­های کنترل و داده­ی پیش­بینی شده قرار می­گیرد. در آینده محک­های بیشتری به مجموعه­ی محک Mibench اضافه خواهد شد. محک­های صنعتی و اتوماتیک آینده شامل نرم­افزار مدولاسیون عرض پالس (PWM) شبیه سازی محیط مجازی و یک برنامه­ی زمان­بندی سیستم عامل real-time می­شوند. محک­های شبکه جدید نیز شامل پیوند[[64]](#footnote-64) جریان بسته­های TCP/IP و سایر دستکاری­ها روی بسته­ها می­باشد.

1. مروری بر كارهای مرتبط
   1. مقدمه

همانطور که پیش­تر گفته شد، يكی از پارامترهای مهم در طراحی سيستم­های تعبيه شده ميزان توان مصرفي در آن­هاست. در اين پروژه نیز هدف بر اين است كه برنامه­های بسته محک MiBench كه يک بسته استاندارد برای كاربردهای تعبيه شده است بر روی يک سيستم تعبيه شده مبتني بر ARM اجرا شده و ميزان توان مصرفی آن­ها به ازای قسمت­های مختلف برنامه استخراج گرديده و تحليل شود. برای این منظور از شبیه­سازی به نام MEET استفاده کردیم. این برنامه مبتنی بر Sim-profile که یک بخشی از مجموعه­ی شبیه­ساز Simplescalar است، می­باشد.

مجموعه ابزار Simplescalar یک زیربنای نرم افزاری سیستم است که در مدل کردن برنامه­ها برای تجزیه و تحلیل عملکرد برنامه­ها، مدل­سازی دقیق میکرومعماری­ها و تاییدیه شرکت­های نرم­افزاری- سخت­افزاری استفاده می­شود. با استفاده از ابزارهای Simplescalar کاربران می­توانند برنامه­های واقعی را که قرار است بر روی طیف وسیعی از سیستم­ها و پردازنده­های مدرن اجرا شوند، را شبیه­سازی کنند.

در ادامه­ی این فصل به توضیحی در ارتباط با شبیه­ساز MEET، سیستم مورد نیاز برای اجرای آن و نحوه­ی استفاده از آن می­پردازیم.

* 1. آشنایی با شبیه­ساز MEET[[65]](#footnote-65)
     1. درباره­ی MEET

MEET یک ابزار اندازه­گیری انرژی مبتنی بر متن برای میکروکنترلرهای AT91SAM7x256 است که توسط آقای مصطفی بزاز در آزمایشگاه سیستم­های تعبیه شده در دانشگاه صنعتی شریف ایران توسعه یافته است[3]. مدل انرژی استفاده شده در MEET، مدل ارائه شده در مقاله­ تحت عنوان "مدل و ابزار تخمین دقیق انرژی در سطح دستورالعمل برای سیستم­های تعبیه شده"[[66]](#footnote-66) که برای انتشار در مقالات IEEE پذیرفته شده است، می­باشد. این مدل انرژی مصرفی پردازنده، حافظه­ی رم استاتیک، حافظه­ی فلش و کنترلر حافظه را تخمین می­زند و لوازم جانبی دیگر مانند RS232 را شامل نمی­شود. بنابراین شبیه­ساز MEET نمی­تواند انرژی مصرفی عبارات Printf و یا دستورالعمل­های ورودی/خروجی را تخمین بزند.

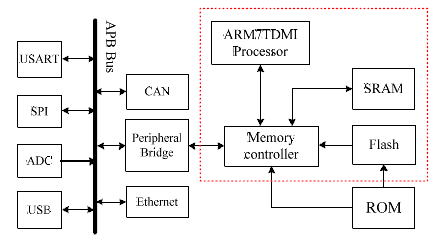
این برنامه مبتنی بر Sim-profile که یک بخشی از مجموعه­ی شبیه­ساز Simplescalar است، می­باشد[2]. MEET یک تصویر باینری سازگار با ARM7TDMI را می­گیرد و برنامه را در سطح دستورالعمل شبیه­سازی می­کند. کار با MEET بسیار شبیه به Sim-profile است و هرکسی با دانش اندکی از نحوه­ی ساخت و اجرای پروژه در شبیه­ساز Simplescalar، برای کار با MEET مشکلی نخواهد داشت. برای ساده کردن فرآیند کامپایل و اجرای برنامه­ها، سه پوسته­ی اسکریپت به بسته اضافه شده­اند: build.sh، extractor.sh و run.sh. build.sh یک پوشه برای فراخوانی کامپایلر Sourcery CodeBench با گزینه­های مناسب است. extractor.sh برای تحلیل برنامه­ی باینری و یافتن آدرس نقطه­ی شروع برنامه استفاده می­شود و run.sh نیز یک پوشه برای فراخوانی MEET با گزینه­های مناسب است. همان­طور که قبلا ذکر شد، MEET نمی­تواند انرژی مصرفی RS232 را تخمین بزند. بنابراین برنامه­ی شبیه سازی شده نمی­تواند شامل هر نوع عبارات خروجی باشد. ساده­ترین راه برای حذف تمام عبارات printf، اضافه کردن یک دستور برای تغییر تعریف printf به یک دستور بی­ارزش است. ( به عنوان مثال، بعد از تمام شدن دستورات include، یک دستور #define printf اضافه شود).

MEET توانایی پروفایل Sim-profile، که می­تواند با قابلیت برآورد انرژی برای ایجاد یک ابزار پروفایل انرژی ترکیب شود را به ارث برده است. Sim-profile می­تواند یک برنامه را در برابر اندازه­های داده شده پروفایل کند که می­تواند یک متغیری باشد که انرژی مصرفی کل برنامه را نگه داشته است. خروجی انرژی مصرفی به ازای هر دستورالعمل است که می­تواند به شناسایی نقاط برنامه کمک کند.

لازم به ذکر است که MEET برخی از محدودیت­های نسخه­ی سیمپل­اسکالر ARM را نیز به ارث می­برد. دستورالعمل­های خاصی از ARM ISA در نسخه­ی سیمپل­اسکالر ARM، مانند دستورات SWP، MSR، MRS و SWI قابل پیاده­سازی نیستند. به عنوان یک نتیجه، MEET نمی­تواند برنامه­هایی را که به این دستورالعمل­ها وابسته هستند، مانند کرنل لینوکس، شبیه­سازی کند. امید است که این مشکل در نسخه­های بعدی MEET اصلاح شود.

* + 1. بستر سخت­افزاری

بستر سخت­افزاری شبیه­سازی شده توسط MEET، یک میکروکنترلر AT91SAM7X256 است که با 64 کیلوبایت حافظه­ی رم استاتیک، 256 کیلوبایت حافظه فلش و یک پردازنده­ی ARM7TDMI مجهز شده است. ساختار داخلی این میکروکنترلر در شکل 3-1 زیر نشان داده شده است. مدل برآورد انرژی شامل انرژی مصرفی هسته­ی پردازنده، رم استاتیک و فلش است. حافظه­ی فلش جهت ذخیره­سازی کد و داده­های فقط خواندنی استفاده می­شود، در حالی که حافظه­ی رم استاتیک به عنوان داده­های زمان اجرا مورد استفاده قرار می­گیرند.



بستر سخت­افزاری شبیه­سازی شده توسط MEET

MEET بین دسترسی به حافظه­ی فلش و دسترسی به حافظه­ی رم استاتیک به وسیله­ی مقدار آدرس هدف تمایز ایجاد می­کند. حافظه­ی فلش از آدرس 0x100000 شروع می­شود در حالی­که حافظه­ی رم استاتیک از آدرس 0x200000 شروع می­شود. بنابراین برای رسیدن به نتایج دقیق، باید اسکریپت لینکر را با توجه به تنظیمات حافظه تغییر داد.

* + 1. سیستم مورد نیاز

سیستم مورد نیاز برای MEET بسیار شبیه به سیمپل­اسکالر است. MEET را می­توان به وسیله­ی کامپایلر GCC در محیط لینوکس کامپایل کرد. ورژن 1.1 با استفاده از تنظیمات زیر آزمایش شده است، اما ساخت آن با استفاده از ورژن­های دیگر لینوکس و GCC باید آسان باشد.

* Ubuntu 12.04 32 bit + GCC 4.6.3
* Ubuntu 10.04 32 bit + GCC 4.4.1
  + 1. استفاده از MEET

فرض کنید که شبیه­ساز MEET و نسخه­ی آرشیو Sourcery CodeBench بر روی سیستم شما نصب است، شما می­توانید برنامه­ی خود را با استفاده از build.sh بسازید و آن را با استفاده از run.sh شبیه­سازی کنید.

اطمینان حاصل کنید که برنامه­ی شما شامل هیچ یک از دستورات چاپ نباشد. (به عنوان مثال دستور #define printf() را بعد از دستورات include اضافه کنید، مانند قطعه کد زیر).

#include <stdio.h>

#define LENGTH 2000

#ifdef NO\_PRINT

#define printf(S,...)

#endif

فولدر جاری خود را به فولدر MEET (MEETfolder) تغییر دهید (به عنوان مثال مسیر فولدر MEET ما به صورت /home/mostafa/Desktop/MEET می­باشد). این مرحله فقط در صورتی که می­خواهید از build.sh و run.sh استفاده کنید لازم است.

* cd /home/mostafa/Desktop/MEET

با استفاده از اسکریپت build.sh کد سورس خود را کامپایل کنید (به عنوان مثال /home/mostafa/source/quicksort.c ).

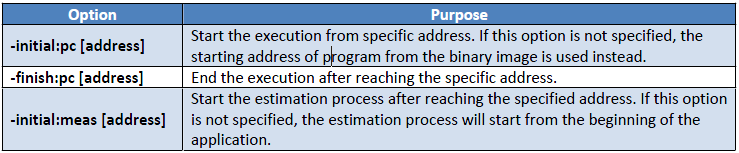
* sh build.sh /home/mostafa/source/quicksort.c

/home/mostafa/source/quicksort

برنامه را شبیه­سازی کنید (فرض کنید می­خواهیم انرژی مصرفی کل برنامه که شامل مراحل مقداردهی­های اولیه هم می­باشد را برآورد کنیم).

* sh run.sh /home/mostafa/source/quicksort main main
  + 1. گزینه­ها[[67]](#footnote-67)

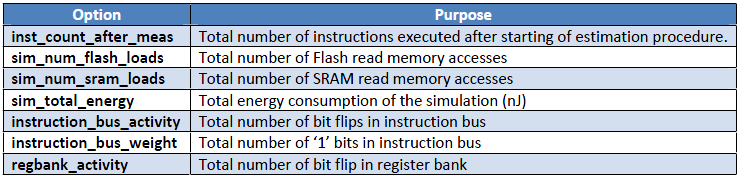
اجرای MEET بدون هیچ آرگومانی، تمامی گزینه­های برنامه را به همراه توضیحات هر گزینه چاپ می­کند. بیشتر این گزینه­ها شبیه به گزینه­های موجود در Sim-profile است. تنها 3 آپشن جدید وجود دارد که در جدول زیر لیست شده است.



گزینه­های جدید MEET

* + 1. مشخصه­های شبیه­سازی[[68]](#footnote-68)

مشابه قسمت قبل، نتایج نهایی شبیه­سازی در MEET به جز در 7 مورد شبیه Sim-profile است. این 7 مورد در جدول زیر لیست شده­اند.



مشخصه­های جدید شبیه­سازی

* + 1. شرح نتایج شبیه­ساز MEET

در ادامه به توضیح هدف برخی از نتایجی که از شبیه­ساز MEET به دست می­آید می­پردازیم.

* sim\_num\_insn: تعداد کل دستورالعمل­های اجرا شده
* inst\_count\_after\_meas: تعداد کل دستورالعمل­های اجرا شده پس از اندازه­گیری
* sim\_num\_refs: تعداد دفعات کل بارگیری و ذخیره­سازی­های اجرا شده
* sim\_num\_loads: تعداد کل دفعات دسترسی به حافظه­ (برای خواندن)
* sim\_num\_flash\_loads: تعداد کل دفعات دسترسی به حافظه­ی فلش (برای خواندن)
* sim\_num\_sram\_loads: تعداد کل دفعات دسترسی به حافظه­ی رم استاتیک (برای خواندن)
* sim\_num\_stores: تعداد کل دفعات دسترسی به حافظه­ (برای نوشتن)
* sim\_total\_energy: انرژی مصرفی کل (بر حسب نانو ژول)
* sim\_elapsed\_time: کل زمان شبیه­سازی (بر حسب ثانیه)
* sim\_inst\_rate: سرعت شبیه­سازی (بر حسب تعداد دستورالعمل بر ثانیه)
* instruction\_bus\_activity: تعداد کل دستکاری­های بیتی در گذرگاه دستورالعمل
* instruction\_bus\_weight: تعداد کل بیت­های "1" در گذرگاه دستورالعمل
* regbank\_activity: تعداد کل دستکاری­های بیتی در رجیستر بانک
* ld\_text\_base: آدرس بخش پایه در متن کد برنامه
* ld\_text\_bound: آدرس بخش مرزی در متن کد برنامه
* ld\_prog\_entry: نقطه­ی ورود به برنامه ( مقدار اولیه­ی PC)
* mem.page\_count: تعداد کل صفحات اختصاص داده شده
* mem.page\_mem: اندازه کل صفحات حافظه اختصاص داده شده

1. شبیه ساز MEET
   1. مقدمه

در این بخش به شرح نتایجی که از شبیه­سازی الگوریتم­های بسته­ی محک Mibench در طی این پروژه در شبیه­ساز MEET به دست آمده و کارهایی که در این راستا انجام شده است، پرداخته می­شود.

همان­طور که می­دانید، اعمال ورودی به برخی سیستم­های سخت افزاری از طریق حافظه­های جانبی ساده، از جمله میکروکنترلی که در آزمایشگاه ما استفاده می­شود (AT91SAM7x256)، امکان پذیر نیست. بنابراین در شبیه­ساز MEET که برنامه­ها را برای استفاده بر روی این میکروکنترلر شبیه­سازی می­کند، نیز همین مسئله وجود دارد و بنابراین برای این که کدها قابل شبیه سازی باشند، باید ابتدا برنامه­ها را به صورت هاردکد[[69]](#footnote-69) درآوریم. هاردکد کردن یک عمل توسعه نرم افزار است که در آن به جای آنکه اطلاعات ورودی از یک منبع خارجی دریافت شود و یا اینکه داده در درون برنامه تولید شود و در خود برنامه قالب­بندی شود، اطلاعات ورودی برنامه مستقیماً در درون کد برنامه قرار می­گیرد و یا اینکه یک قالب­بندی ثابت برای آن­ها وجود دارد. برای هاردکد کردن هر یک از برنامه­ها لازم بود که ابتدا الگوریتم آن را بررسی کنیم و سپس از نحوه­ی ورودی گرفتن آن اطلاع یابیم.

مسئله­ی دیگری که پس از هاردکد کردن وجود داشت، این بود که اکثر برنامه‌­هایی که قصد کامپایل آن را داشتیم در چند فایل تعریف شده بود و توابع آن در فایل دیگری بودند، در حالی­که اسکریپت همراه نرم‌افزار MEET فقط برای برنامه‌های تک فایله کاربرد دارد. در نتیجه دو راه برای برطرف کردن این مشکل وجود داشت، یکی اینکه باید در متن اسکریپت MEET، نحوه‌ی کامپایل یک برنامه را بررسی می­کردیم و سپس خودمان دستور کامپایل برنامه را می­ساختیم. راه ساده­تر دیگر این بود که تمامی فایل­های برنامه را در یک فایل ادغام کرده و سپس شبیه­سازی را انجام دهیم. به دلیل سختی روش اول، راه دوم را برگزیدیم.

در ادامه به شرح هر یک از الگوریتم­های شبیه­سازی شده و فرمت ورودی گرفتن آن­ها می­پردازیم. سپس به نتایج حاصل از شبیه سازی می­پردازیم.

* 1. شرح برنامه­ها و نتایج شبیه­سازی آن­ها
     1. Bitcount

همان­طور که پیش از این گفته شد، این الگوریتم قابلیت دستکاری بیت­ها در یک پردازنده را به وسیله­ی شمارش تعداد بیت­ها در یک آرایه از اعداد صحیح تست می­کند. این فرآیند از طریق پنج روش قابل انجام است:

1. شمارنده­ی بهینه­شده­ی یک بیت در هر حلقه[[70]](#footnote-70)
2. شمارش بازگشتی 4 بیتی[[71]](#footnote-71)
3. شمارش غیر بازگشتی 4 بیتی با استفاده از جدول جستجو[[72]](#footnote-72)
4. شمارش غیر بازگشتی 8 بیتی با استفاده از جدول جستجو[[73]](#footnote-73)
5. شیفت دادن و شمارش بیت­ها[[74]](#footnote-74)

در الگوریتم شماره­ی 1 (شمارنده­ی بهینه­شده­ی یک بیت در هر حلقه)، حلقه­ی loop به ازای هر بیت از مجموعه­ی x، یک بار اجرا می­شود. به طور متوسط سرعت اجرای آن دو برابر الگوریتم شیفت و تست است.

الگوریتم شماره 2 که توسط رتکو تامیک[[75]](#footnote-75) نوشته شده است، به صورت بازگشتی شمارش بیت­ها را به صورت 4 بیتی انجام می­دهد.

الگوریتم شماره 3 که توسط باب استوت[[76]](#footnote-76) نوشته شده است، با استفاده از یک جدول جستجو شمارش غیر بازگشتی را انجام می­دهد. نکته­ای که وجود دارد این است که 16 ورودی اول جدول برای این الگوریتم استفاده می­شود. مابقی می­توانند حذف شوند.

الگوریتم شماره 4 که توسط بروس ودینگ و اوک ریتسما[[77]](#footnote-77) نوشته شده است، همانند الگوریتم 3 است، با این تفاوت که شمارش به صورت 8 بیتی انجام می­شود.

در الگوریتم شماره 5 نیز شمارش در بیت­ها با شیفت دادن بیت­ها انجام می­شود.

* ورودی الگوریتم:

مجموعه داده­های ورودی برای این الگوریتم یک آرایه از اعداد صحیح با مقادیر ‘0’ و ‘1’ است.

* نتایج شبیه­سازی:

نتایج شبیه­سازی برای الگوریتم Bitcount به شرح زیر است:

نتایج شبیه­سازی برای الگوریتم Bitcount

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 7635508667 # total number of instructions executed |
| inst\_count\_after\_meas | 7635508666 # total number of instructions executed after measurement |
| sim\_num\_refs | 4270896736 # total number of loads and stores executed |
| sim\_num\_loads | 3096829403 # total number of read memory accesses |
| sim\_num\_flash\_loads | 396365009 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 2700464394 # total number of SRAM read memory accesses |
| sim\_num\_stores | 1350235259 # total number of write memory accesses |
| sim\_total\_energy | 134220032.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 16046 # total simulation time in seconds |
| sim\_inst\_rate | 475851.2194 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 70505772848 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 91255679090 # total number of 1 count in instruction bus |
| regbank\_activity | 37597991434 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x0010be10 # program text (code) segment bound |
| ld\_text\_size | 48656 # program text (code) size in bytes |
| ld\_data\_base | 0x0010b978 # program initialized data segment base |
| ld\_data\_bound | 0x00200b20 # program initialized data segment bound |
| ld\_data\_size | 1003944 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 15 # total number of pages allocated |
| mem.page\_mem | 60k # total size of memory pages allocated |
| mem.ptab\_misses | 15 # total first level page table misses |
| mem.ptab\_accesses | 24165254594 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0000 # first level page table miss rate |

* + 1. دیکسترا

این الگوریتم یکی از [الگوریتم‌های پیمایش گراف](http://fa.wikipedia.org/wiki/%D8%A7%D9%84%DA%AF%D9%88%D8%B1%DB%8C%D8%AA%D9%85%E2%80%8C%D9%87%D8%A7%DB%8C_%D9%BE%DB%8C%D9%85%D8%A7%DB%8C%D8%B4_%DA%AF%D8%B1%D8%A7%D9%81) است که [مسئله­ی کوتاه‌ترین مسیر](http://fa.wikipedia.org/w/index.php?title=%D9%85%D8%B3%D8%A6%D9%84%D9%87%D9%94_%DA%A9%D9%88%D8%AA%D8%A7%D9%87%E2%80%8C%D8%AA%D8%B1%DB%8C%D9%86_%D9%85%D8%B3%DB%8C%D8%B1&action=edit&redlink=1&preload=%D8%A7%D9%84%DA%AF%D9%88:%D8%A7%DB%8C%D8%AC%D8%A7%D8%AF+%D9%85%D9%82%D8%A7%D9%84%D9%87/%D8%A7%D8%B3%D8%AA%D8%AE%D9%88%D8%A7%D9%86%E2%80%8C%D8%A8%D9%86%D8%AF%DB%8C&editintro=%D8%A7%D9%84%DA%AF%D9%88:%D8%A7%DB%8C%D8%AC%D8%A7%D8%AF+%D9%85%D9%82%D8%A7%D9%84%D9%87/%D8%A7%D8%AF%DB%8C%D8%AA%E2%80%8C%D9%86%D9%88%D8%AA%DB%8C%D8%B3&summary=%D8%A7%DB%8C%D8%AC%D8%A7%D8%AF+%DB%8C%DA%A9+%D9%85%D9%82%D8%A7%D9%84%D9%87+%D9%86%D9%88+%D8%A7%D8%B2+%D8%B7%D8%B1%DB%8C%D9%82+%D8%A7%DB%8C%D8%AC%D8%A7%D8%AF%DA%AF%D8%B1&nosummary=&prefix=&minor=&create=%D8%AF%D8%B1%D8%B3%D8%AA+%DA%A9%D8%B1%D8%AF%D9%86+%D9%85%D9%82%D8%A7%D9%84%D9%87+%D8%AC%D8%AF%DB%8C%D8%AF) از مبدأ واحد را برای [گراف‌های وزن‌داری](http://fa.wikipedia.org/w/index.php?title=%DA%AF%D8%B1%D8%A7%D9%81_%D9%88%D8%B2%D9%86%E2%80%8C%D8%AF%D8%A7%D8%B1&action=edit&redlink=1&preload=%D8%A7%D9%84%DA%AF%D9%88:%D8%A7%DB%8C%D8%AC%D8%A7%D8%AF+%D9%85%D9%82%D8%A7%D9%84%D9%87/%D8%A7%D8%B3%D8%AA%D8%AE%D9%88%D8%A7%D9%86%E2%80%8C%D8%A8%D9%86%D8%AF%DB%8C&editintro=%D8%A7%D9%84%DA%AF%D9%88:%D8%A7%DB%8C%D8%AC%D8%A7%D8%AF+%D9%85%D9%82%D8%A7%D9%84%D9%87/%D8%A7%D8%AF%DB%8C%D8%AA%E2%80%8C%D9%86%D9%88%D8%AA%DB%8C%D8%B3&summary=%D8%A7%DB%8C%D8%AC%D8%A7%D8%AF+%DB%8C%DA%A9+%D9%85%D9%82%D8%A7%D9%84%D9%87+%D9%86%D9%88+%D8%A7%D8%B2+%D8%B7%D8%B1%DB%8C%D9%82+%D8%A7%DB%8C%D8%AC%D8%A7%D8%AF%DA%AF%D8%B1&nosummary=&prefix=&minor=&create=%D8%AF%D8%B1%D8%B3%D8%AA+%DA%A9%D8%B1%D8%AF%D9%86+%D9%85%D9%82%D8%A7%D9%84%D9%87+%D8%AC%D8%AF%DB%8C%D8%AF) که یال با وزن منفی ندارند، حل می‌کند و در نهایت با ایجاد [درخت کوتاه‌ترین مسیر](http://fa.wikipedia.org/wiki/%D8%AF%D8%B1%D8%AE%D8%AA_%DA%A9%D9%88%D8%AA%D8%A7%D9%87%E2%80%8C%D8%AA%D8%B1%DB%8C%D9%86_%D9%85%D8%B3%DB%8C%D8%B1)، کوتاه‌ترین مسیر از مبدأ به همه­ی [رأس‌های](http://fa.wikipedia.org/wiki/%D8%B1%D8%A3%D8%B3_(%D9%86%D8%B8%D8%B1%DB%8C%D9%87_%DA%AF%D8%B1%D8%A7%D9%81)) [گراف](http://fa.wikipedia.org/wiki/%DA%AF%D8%B1%D8%A7%D9%81) را به دست می‌دهد. همچنین می‌توان از این الگوریتم برای پیدا کردن کوتاه‌ترین مسیر از مبدأ تا رأس مقصد به این ترتیب بهره جست که در حین اجرای الگوریتم به محض پیدا شدن کوتاه‌ترین مسیر از مبدأ به مقصد، الگوریتم را متوقف کرد.

نام این الگوریتم بر اساس نام ارائه‌دهنده هلندی آن، یعنی [اِدسخِر دِیکسترا](http://fa.wikipedia.org/wiki/%D8%A7%D8%AF%D8%B3%D8%AE%D8%B1_%D8%AF%DB%8C%DA%A9%D8%B3%D8%AA%D8%B1%D8%A7) انتخاب شده است.

روند الگوریتم دیکسترا مطابق زیر می­باشد :

1. انتخاب راس مبدا
2. مجموعه­ی S ، شامل رئوس گراف ، معین می­شود. در شروع، این مجموعه تهی بوده و با پیشرفت الگوریتم، این مجموعه رئوسی که کوتاه ترین مسیر به آن ها یافت شده است را در بر می­گیرد.
3. راس مبدا با اندیس صفر را در داخل S قرار می­دهد.
4. برای رئوس خارج از S ، اندیسی معادل "طول یال + اندیس" راس قبلی را در نظر می­گیرد . اگر راس خارج از مجموعه دارای اندیس باشد، اندیس جدید کمترین مقدار از بین اندیس قبلی و "طول یال + اندیس" راس قبلی می­باشد.
5. از رئوس خارج مجموعه، راسی با کمترین اندیس انتخاب شده و به مجموعه­ی S اضافه می­گردد.
6. این کار را دوباره از مرحله­ی 4 ادامه داده تا راس مقصد وارد مجموعه­ی S شود.

در پایان اگر راس مقصد دارای اندیس باشد، اندیس آن نشان دهنده­ی مسافت بین مبدا و مقصد می­باشد. در غیر این صورت هیچ مسیری بین مبدا و مقصد موجود نمی‌باشد.

همچنین برای پیدا کردن مسیر می­توان اندیس دیگری برای هر راس در نظر گرفت که نشان دهنده ی راس قبلی در مسیر طی شده باشد. بدین ترتیب پس از پایان اجرای الگوریتم، با دنبال کردن رئوس قبلی از مقصد به مبدا، کوتاه ترین مسیر بین دو نقطه نیز یافت می­شود.

* ورودی الگوریتم:

مجموعه داده­های ورودی در این الگوریتم ماتریس مجاورت گراف است.

* نتایج شبیه­سازی:

نتایج شبیه­سازی برای الگوریتم Dijkstra به شرح زیر است:

برای ورودی آرایه­ای با ابعاد 50×50:

نتایج شبیه­سازی الگوریتم Dijkstra، برای ورودی با سایز کوچک

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 121265791 # total number of instructions executed |
| inst\_count\_after\_meas | 121265790 # total number of instructions executed after measurement |
| sim\_num\_refs | 38196394 # total number of loads and stores executed |
| sim\_num\_loads | 35781558 # total number of read memory accesses |
| sim\_num\_flash\_loads | 23922502 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 11859056 # total number of SRAM read memory accesses |
| sim\_num\_stores | 3039751 # total number of write memory accesses |
| sim\_total\_energy | 134447904.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 141 # total simulation time in seconds |
| sim\_inst\_rate | 860041.0709 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 1472199374 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 1355662481 # total number of 1 count in instruction bus |
| regbank\_activity | 756661700 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x00116898 # program text (code) segment bound |
| ld\_text\_size | 92312 # program text (code) size in bytes |
| ld\_data\_base | 0x001147a8 # program initialized data segment base |
| ld\_data\_bound | 0x002033d0 # program initialized data segment bound |
| ld\_data\_size | 977960 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 30 # total number of pages allocated |
| mem.page\_mem | 120k # total size of memory pages allocated |
| mem.ptab\_misses | 262174 # total first level page table misses |
| mem.ptab\_accesses | 319992708 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0008 # first level page table miss rate |

برای ورودی آرایه­ای با ابعاد 100×100:

نتایج شبیه­سازی الگوریتم Dijkstra، برای ورودی با سایز بزرگ

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 1726339312 # total number of instructions executed |
| inst\_count\_after\_meas | 1726339311 # total number of instructions executed after measurement |
| sim\_num\_refs | 496383722 # total number of loads and stores executed |
| sim\_num\_loads | 469821885 # total number of read memory accesses |
| sim\_num\_flash\_loads | 335181462 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 134640423 # total number of SRAM read memory accesses |
| sim\_num\_stores | 29928889 # total number of write memory accesses |
| sim\_total\_energy | 135834336.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 2254 # total simulation time in seconds |
| sim\_inst\_rate | 765900.3159 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 21613325806 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 19185618788 # total number of 1 count in instruction bus |
| regbank\_activity | 11418349119 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x0011bdb0 # program text (code) segment bound |
| ld\_text\_size | 114096 # program text (code) size in bytes |
| ld\_data\_base | 0x001147a8 # program initialized data segment base |
| ld\_data\_bound | 0x0020aa90 # program initialized data segment bound |
| ld\_data\_size | 1008360 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 43 # total number of pages allocated |
| mem.page\_mem | 172k # total size of memory pages allocated |
| mem.ptab\_misses | 262187 # total first level page table misses |
| mem.ptab\_accesses | 4451494474 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0001 # first level page table miss rate |

* + 1. Stringsearch

همان­طور که پیش از این گفته شده بود، این محک با استفاده از الگوریتم مقایسه­ی حساس به حروف[[78]](#footnote-78) در عبارات به دنبال کلمات داده شده می­گردد.

این الگوریتم در واقع بخشی از الگوریتم جستجوی رشته­ی پرات- بویر- مور[[79]](#footnote-79) است که در سال 1991 توسط جری کافین نوشته شده است. این بخش از الگوریتم که در Mibench استفاده شده، در اوایل سال 1991 از برنامه ی اصلی جدا شد و با بازنویسی جداگانه­ی آن، برای استفاده­ی عمومی آماده شد. سپس در اواخر مارس و اوایل آپریل با کمک تاد اسمیت[[80]](#footnote-80) اصلاح شد.

در این الگوریتم تابع Init\_search به همراه رشته ای که شامل کلمه­ی مورد نظر برای جستجو است، برای قرار گرفتن در جدول مقدار اولیه فراخوانی می­شود. سپس تابع Strsearch به همراه یک بافر برای عملیات جستجو صدا زده می­شود.

* ورودی الگوریتم

ورودی این الگوریتم شامل دو رشته می­باشد که یکی از آن­ها متنی است که در آن به دنبال کلمات مورد نظر هستیم و دیگری همان کلمات مورد نظر است.

* نتایج شبیه­سازی

نتایج شبیه­سازی برای الگوریتم Stringsearch به شرح زیر است:

برای ورودی کوچک با سایز تقریبی 60 رشته:

نتایج شبیه­سازی الگوریتم Stringsearch، برای ورودی با سایز کوچک

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 445431 # total number of instructions executed |
| inst\_count\_after\_meas | 445430 # total number of instructions executed after measurement |
| sim\_num\_refs | 125710 # total number of loads and stores executed |
| sim\_num\_loads | 95276 # total number of read memory accesses |
| sim\_num\_flash\_loads | 32390 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 62886 # total number of SRAM read memory accesses |
| sim\_num\_stores | 31159 # total number of write memory accesses |
| sim\_total\_energy | 856388.8125 # total energy consumption (nJ) |
| sim\_elapsed\_time | 1 # total simulation time in seconds |
| sim\_inst\_rate | 445431.0000 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 1773522 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 2479752 # total number of 1 count in instruction bus |
| regbank\_activity | 373078 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x00101708 # program text (code) segment bound |
| ld\_text\_size | 5896 # program text (code) size in bytes |
| ld\_data\_base | 0x00100ce4 # program initialized data segment base |
| ld\_data\_bound | 0x00200868 # program initialized data segment bound |
| ld\_data\_size | 1047428 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 5 # total number of pages allocated |
| mem.page\_mem | 20k # total size of memory pages allocated |
| mem.ptab\_misses | 262149 # total first level page table misses |
| mem.ptab\_accesses | 901170 # total page table accesses |
| mem.ptab\_miss\_rate | 0.2909 # first level page table miss rate |

برای ورودی بزرگ با سایز تقریبی 1300 رشته:

نتایج شبیه­سازی الگوریتم Stringsearch، برای ورودی با سایز بزرگ

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 4473903 # total number of instructions executed |
| inst\_count\_after\_meas | 4473902 # total number of instructions executed after measurement |
| sim\_num\_refs | 2907554 # total number of loads and stores executed |
| sim\_num\_loads | 2201702 # total number of read memory accesses |
| sim\_num\_flash\_loads | 749602 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 1452100 # total number of SRAM read memory accesses |
| sim\_num\_stores | 718799 # total number of write memory accesses |
| sim\_total\_energy | 12161095.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 9 # total simulation time in seconds |
| sim\_inst\_rate | 497100.3333 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 40533414 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 57045142 # total number of 1 count in instruction bus |
| regbank\_activity | 8565134 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x00104850 # program text (code) segment bound |
| ld\_text\_size | 18512 # program text (code) size in bytes |
| ld\_data\_base | 0x00100d04 # program initialized data segment base |
| ld\_data\_bound | 0x00200868 # program initialized data segment bound |
| ld\_data\_size | 1047396 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 10 # total number of pages allocated |
| mem.page\_mem | 40k # total size of memory pages allocated |
| mem.ptab\_misses | 262154 # total first level page table misses |
| mem.ptab\_accesses | 14566378 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0180 # first level page table miss rate |

* + 1. بلوفیش

این الگوریتم یک رمزنگار بلوکی است که بر روی مقادیر 64 بیتی (8بایتی) عملیات انجام می­دهد. این الگوریتم از مقادیر مختلف کلید برای رمزنگاری می­تواند استفاده کند ولی عموما مقدار 128 بیتی (16بایتی) برای آن در نظر گرفته می­شود. این الگوریتم می­تواند در همه­ مدهایی که DES استفاده می­کند، استفاده شود. کتابخانه­ بلوفیش در مدهای ecb، cbc، cfb64، ofb64 پیاده­سازی می­شود.

الگوریتم شامل دو بخش است: یک بخش بسط کلید و یک بخش رمزگذاری داده. بسط کلید، یک کلید با طول متغیر و حداکثر ۵۶ بایت (۴۴۸بیت) را به آرایه‌ای از چندین زیرکلید مجموعا ۴۱۶۸ بایت، تبدیل می‌کند. بلوفیش 16 دور دارد. هر دور شامل یک جایگشت وابسته به کلید و یک جانشانی وابسته به کلید و داده‌است. تمامی عملگرها [XORها](http://fa.wikipedia.org/wiki/%DB%8C%D8%A7%DB%8C_%D9%85%D8%A7%D9%86%D8%B9%D8%A9%D8%A7%D9%84%D8%AC%D9%85%D8%B9) و جمع‌هایی هستند که بر روی کلمات ۳۲ بیتی اعمال می‌شوند. تنها عملگر اضافی چهار آرایه شاخص­دار جستجوی داده در هر دور است. طول قطعه در بلوفیش 64 بیت و طول کلید از ۳۲ بیت تا ۴۴۸ بیت متغیر است.

بلوفیش کمی سریع­تر از DES و بسیار سریع­تر از IDEA و RC2 است. در کل می­توان بلوفیش را یکی از سریع­ترین رمزنگارهای بلوکی دانست.

کلیه­ی توابع رمزنگاری آرگومانی را با عنوان BF\_KEY استفاده می­کنند. BF\_KEY یک شکل بسط یافته از کلید رمزنگاری بلوفیش است. در همه­ی مدهای الگوریتم بلوفیش، در الگوریتم­های رمزگشایی نیز BF\_KEY همانند الگوریتم­های رمزنگاری استفاده می­شود.

تعاریف BF\_ENCRYPT و BF\_DECRYPT نیز برای مشخص کردن حالت رمزنگاری یا رمزگشایی برای توابعی که از پرچم رمزنگاری/رمزگشایی استفاده می­کنند، استفاده می­شود.

در Mibench به دلایل زیر تنها مدهای ecb، cbc، cfb64، ofb64 پیاده­سازی شده است:

* Ecb یک رمزنگار پایه­ی بلوفیش است.
* Cbc یک فرم نرمال زنجیره­ای برای رمزنگارهای بلوکی است.
* cfb64 می­تواند برای رمزنگارهای تک کاراکتری استفاده شود، بنابراین نیازی نیست که ورودی و خروجی مضربی از 8 باشد.
* ofb64 مشابه cfb64 ولی بیشتر شبیه رمزنگارهای جریانی است. به آن اندازه امن نیست ولی دیگر نیازی به مد رمزنگاری/رمزگشایی ندارد.
* ورودی الگوریتم:

مجموعه داده­های ورودی در این الگوریتم یک فایل متنی ASCII بزرگ و کوچک از یک مقاله­ی آنلاین است.

* نتایج شبیه­سازی:

نتایج شبیه­سازی برای الگوریتم رمزنگاری بلوفیش به شرح زیر است:

برای ورودی کوچک با سایز تقریبی 230000 کاراکتر:

نتایج شبیه­سازی الگوریتم بلوفیش، برای ورودی با سایز کوچک

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 1719563275 # total number of instructions executed |
| inst\_count\_after\_meas | 1719563274 # total number of instructions executed after measurement |
| sim\_num\_refs | 665475783 # total number of loads and stores executed |
| sim\_num\_loads | 545080040 # total number of read memory accesses |
| sim\_num\_flash\_loads | 37619855 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 507460185 # total number of SRAM read memory accesses |
| sim\_num\_stores | 209045785 # total number of write memory accesses |
| sim\_total\_energy | 134217744.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 2871 # total simulation time in seconds |
| sim\_inst\_rate | 598942.2762 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 16231957316 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 19249358625 # total number of 1 count in instruction bus |
| regbank\_activity | 16467021175 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x0013e4b0 # program text (code) segment bound |
| ld\_text\_size | 255152 # program text (code) size in bytes |
| ld\_data\_base | 0x00105524 # program initialized data segment base |
| ld\_data\_bound | 0x002019a0 # program initialized data segment bound |
| ld\_data\_size | 1033340 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 124 # total number of pages allocated |
| mem.page\_mem | 496k # total size of memory pages allocated |
| mem.ptab\_misses | 262268 # total first level page table misses |
| mem.ptab\_accesses | 4947187970 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0001 # first level page table miss rate |

برای ورودی بزرگ با سایز تقریبی 460000 کاراکتر:

نتایج شبیه­سازی الگوریتم بلوفیش، برای ورودی با سایز بزرگ

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 3701069514 # total number of instructions executed |
| inst\_count\_after\_meas | 3701069513 # total number of instructions executed after measurement |
| sim\_num\_refs | 1431567192 # total number of loads and stores executed |
| sim\_num\_loads | 1172005864 # total number of read memory accesses |
| sim\_num\_flash\_loads | 81112019 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 1090893845 # total number of SRAM read memory accesses |
| sim\_num\_stores | 450737251 # total number of write memory accesses |
| sim\_total\_energy | 134217744.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 5643 # total simulation time in seconds |
| sim\_inst\_rate | 655869.1324 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 35021682106 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 41556536267 # total number of 1 count in instruction bus |
| regbank\_activity | 35457766120 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x0017fee8 # program text (code) segment bound |
| ld\_text\_size | 524008 # program text (code) size in bytes |
| ld\_data\_base | 0x001057ec # program initialized data segment base |
| ld\_data\_bound | 0x002019a0 # program initialized data segment bound |
| ld\_data\_size | 1032628 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 255 # total number of pages allocated |
| mem.page\_mem | 1020k # total size of memory pages allocated |
| mem.ptab\_misses | 262399 # total first level page table misses |
| mem.ptab\_accesses | 10647436454 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0000 # first level page table miss rate |

* + 1. Rijndael

الگوریتم Rijndael که یک الگوریتم رمزنگاری با گزینه­های 128، 192 و 256 بیتی از کلید و بلوک­هاست، در حال حاضر بعنوان استاندارد رمز نگاری پیشرفته یا AES شناخته می شود. AES به عنوان یک مشخصه برای رمزنگاری داده های دیجیتال است که توسط دولت ایالات متحده اتخاذ شده است و امروزه بصورت جهانی استفاده می شود. AES جانشین DES می باشد. الگوریتمی که توسط AES توصیف می شود، یک الگوریتم کلید متقارن است. به این معنا که از یک کلید مشابه برای رمز کردن و گشودن اطلاعات استفاده می شود.

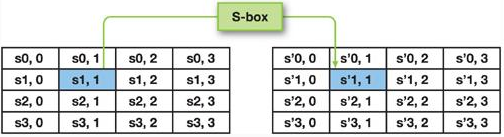
الگوریتم Rijndael توسط دو رمزنویس بلژیکی با نام های جوآن دایمن[[81]](#footnote-81) و وینسنت ریجمن[[82]](#footnote-82) توسعه یافته و به انتخابات AES معرفی شده است و نام آن با تلفیقی از نام مخترعانش به دست آمده است.

در این الگوریتم به جز فایل­هایی که کمتر از دو بلوک دارند، یک بایت از بلوک قبلی، “i” بایت از بلوک فعلی برای رمزنگاری استفاده می­شود و “15-i” نیز به عنوان بافر در نظر گرفته می­شود. برای فایل­هایی که کمتر از دو بلوک (یعنی 0 یا 1 بلوک) دارند، “i+1” برای رمزنگاری و “14-i” به عنوان بافر در نظر گرفته می­شود.

* طرز كار الگوریتم

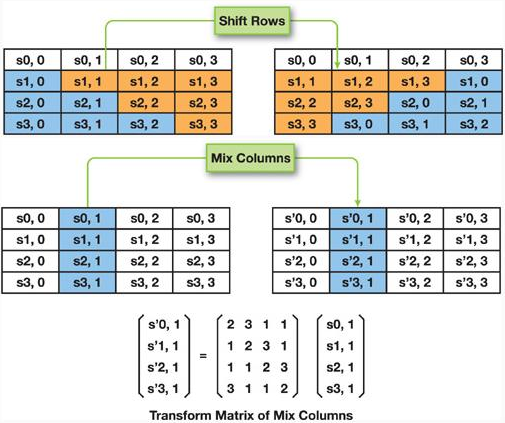
الگوریتم Rijndael بایت به بایت كار می­كند و ورودی اصلی را با كلید رمزنگاری در یك ماتریس 4×4 جفت می­كند. كلید، به طریقی تقسیم یا برنامه­ریزی شده است كه بتواند در مراحل مختلف تكرار به تدریج تزریق شود. اولین قسمت كلید قبل از شروع پروسه 10 مرحله­ای تزریق می­شود. در هر كدام از این مراحل، بایت­ها جابجا می­شوند، ردیف­ها شیفت پیدا می­كنند و ستون­ها تركیب می­شوند.

* **SubBytes**  
  در پروسه جابجایی، بایت­های متن ورودی در یك جعبه جابجایی به نام S-box قرار می­گیرند كه یك ماتریس 16×16 است. هر بایت در یك تقاطع سطر و ستون این ماتریس جا می­گیرد. برای پیدا كردن جای هر بایت اولین عدد صحیح مبنای 16 (nibble) در یك بایت متن اصلی گرفته شده و از آن برای مشخص كردن سطر S-box استفاده می­شود و سپس از دومین nibble برای مشخص كردن ستون استفاده می­شود. كاراكتری كه در تقاطع سطر و ستون انتخاب شده ذخیره می­گردد، به عنوان SubByte برای متن اصلی شناخته می­شود. این پروسه برای هر 16 بایت در ماتریس تكرار می­شود.



* **شیفت ردیف و تركیب ستون­ها**

بایت­هایی كه باید رمزنگاری شوند، توسط جایگذاری تعویض می­شوند وسپس ردیف­ها شیفت پیدا می­كنند. برای مثال اولین ردیف دست نخورده باقی می­ماند، ردیف دوم یك محل به راست جابجا می­شود، سومین ردیف دو محل جابجا می­شود و آخرین ردیف نیز سه محل جابجا می­شود. این پروسه توسط یك فاز تركیب ستون­ها دنبال می­شود كه در آن هر ستون از ماتریس در یك ماتریس دیگر ضرب می­شود تا موقعیت ستون تغییر پیدا كند.



* **كلیدهای Round**

در مرحله بعدی یك كلید Round به هر ستون اضافه می­شود. این كلید در واقع یك تكه كوچك از یك كلید محرمانه است كه برای مراحل بعدی رمزنگاری تزریق می­شود.

* **تكرار**

این تبدیل­ها 9 بار دیگر تكرار می­شوند. در تكرار آخر تركیب ستون ها وجود ندارد و با اضافه كردن كلید Round متن رمزنگاری شده به دست می­آید. كلید نیز به نوبه خود شیفت پیدا می­كند، گرد می­شود و به خودش اضافه می­شود.

* ورودی الگوریتم:

مجموعه داده­های ورودی در این الگوریتم یک فایل متنی ASCII بزرگ و کوچک از یک مقاله­ی آنلاین است.

* نتایج شبیه­سازی:

نتایج شبیه­سازی برای الگوریتم رمزنگاری Rijndael به شرح زیر است:

برای ورودی کوچک با سایز تقریبی 230000 کاراکتر:

نتایج شبیه­سازی الگوریتم Rijndael، برای ورودی با سایز کوچک

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 473197189 # total number of instructions executed |
| inst\_count\_after\_meas | 473197188 # total number of instructions executed after measurement |
| sim\_num\_refs | 194532636 # total number of loads and stores executed |
| sim\_num\_loads | 173797785 # total number of read memory accesses |
| sim\_num\_flash\_loads | 76647842 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 97149943 # total number of SRAM read memory accesses |
| sim\_num\_stores | 27258124 # total number of write memory accesses |
| sim\_total\_energy | 134521840.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 513 # total simulation time in seconds |
| sim\_inst\_rate | 922411.6745 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 5154754102 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 5563857708 # total number of 1 count in instruction bus |
| regbank\_activity | 4970337402 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x00146f88 # program text (code) segment bound |
| ld\_text\_size | 290696 # program text (code) size in bytes |
| ld\_data\_base | 0x00108d80 # program initialized data segment base |
| ld\_data\_bound | 0x00200970 # program initialized data segment bound |
| ld\_data\_size | 1014768 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 75 # total number of pages allocated |
| mem.page\_mem | 300k # total size of memory pages allocated |
| mem.ptab\_misses | 1426049 # total first level page table misses |
| mem.ptab\_accesses | 1345807970 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0011 # first level page table miss rate |

برای ورودی بزرگ با سایز تقریبی 460000 کاراکتر:

نتایج شبیه­سازی الگوریتم Rijndael، برای ورودی با سایز بزرگ

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 944499149 # total number of instructions executed |
| inst\_count\_after\_meas | 944499148 # total number of instructions executed after measurement |
| sim\_num\_refs | 388598372 # total number of loads and stores executed |
| sim\_num\_loads | 347361916 # total number of read memory accesses |
| sim\_num\_flash\_loads | 153062542 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 194299374 # total number of SRAM read memory accesses |
| sim\_num\_stores | 54515898 # total number of write memory accesses |
| sim\_total\_energy | 138249376.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 691 # total simulation time in seconds |
| sim\_inst\_rate | 1366858.3922 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 10298302266 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 11111148228 # total number of 1 count in instruction bus |
| regbank\_activity | 9671043836 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x0017f490 # program text (code) segment bound |
| ld\_text\_size | 521360 # program text (code) size in bytes |
| ld\_data\_base | 0x00108dc0 # program initialized data segment base |
| ld\_data\_bound | 0x00200970 # program initialized data segment bound |
| ld\_data\_size | 1014704 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 132 # total number of pages allocated |
| mem.page\_mem | 528k # total size of memory pages allocated |
| mem.ptab\_misses | 2590958 # total first level page table misses |
| mem.ptab\_accesses | 2687488410 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0010 # first level page table miss rate |

* + 1. Sha

یک الگوریتم درهم سازی امن است که یک پیام خلاصه­ی 160 بیتی را برای یک داده­ی ورودی ایجاد می­کند. این الگوریتم اغلب برای تبادل امن کلیدهای رمزنگاری و برای ایجاد فضاهای دیجیتال ایجاد می­شود. مشخصه‌های اصلی این الگوریتم اولین بار در سال ۱۹۹۳ به عنوان استاندارد درهم سازی ایمن توسط NIST[[83]](#footnote-83) انتشار یافت.

در این الگوریتم تابع sha\_stream، پیام خلاصه را از جریان ورودی محاسبه می­کند. تابع sha\_init پیام خلاصه را مقداردهی اولیه می­کند. تابع sha\_update پس از خواندن هر بایت از ورودی فراخوانده می­شود و پیام خلاصه را به روز می­کند. تابع sha\_final در پس از خوانده شدن کامل ورودی فراخوانده می­شود و به کار محاسبات پیام خلاصه پایان می­دهد. در انتها نیز پیام چاپ می­شود.

* ورودی الگوریتم:

مجموعه داده­های ورودی در این الگوریتم یک فایل متنی ASCII بزرگ و کوچک از یک مقاله­ی آنلاین است.

* نتایج شبیه­سازی:

نتایج شبیه­سازی برای الگوریتم رمزنگاری Sha به شرح زیر است:

برای ورودی کوچک با سایز تقریبی 230000 کاراکتر:

نتایج شبیه­سازی الگوریتم Sha، برای ورودی با سایز کوچک

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 31054210 # total number of instructions executed |
| inst\_count\_after\_meas | 31054209 # total number of instructions executed after measurement |
| sim\_num\_refs | 15440469 # total number of loads and stores executed |
| sim\_num\_loads | 12242005 # total number of read memory accesses |
| sim\_num\_flash\_loads | 2729965 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 9512040 # total number of SRAM read memory accesses |
| sim\_num\_stores | 3228313 # total number of write memory accesses |
| sim\_total\_energy | 43340804.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 64 # total simulation time in seconds |
| sim\_inst\_rate | 485222.0312 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 282604911 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 349660845 # total number of 1 count in instruction bus |
| regbank\_activity | 327609320 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x00144560 # program text (code) segment bound |
| ld\_text\_size | 279904 # program text (code) size in bytes |
| ld\_data\_base | 0x0010b7a4 # program initialized data segment base |
| ld\_data\_bound | 0x002009e0 # program initialized data segment bound |
| ld\_data\_size | 1004092 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 74 # total number of pages allocated |
| mem.page\_mem | 296k # total size of memory pages allocated |
| mem.ptab\_misses | 262218 # total first level page table misses |
| mem.ptab\_accesses | 93356930 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0028 # first level page table miss rate |

برای ورودی بزرگ با سایز تقریبی 460000 کاراکتر:

نتایج شبیه­سازی الگوریتم Sha، برای ورودی با سایز بزرگ

|  |  |
| --- | --- |
| Option | Purpose |
| sim\_num\_insn | 60779163 # total number of instructions executed |
| inst\_count\_after\_meas | 60779162 # total number of instructions executed after measurement |
| sim\_num\_refs | 30348015 # total number of loads and stores executed |
| sim\_num\_loads | 23965532 # total number of read memory accesses |
| sim\_num\_flash\_loads | 4984100 # total number of Flash read memory accesses |
| sim\_num\_sram\_loads | 18981432 # total number of SRAM read memory accesses |
| sim\_num\_stores | 6441545 # total number of write memory accesses |
| sim\_total\_energy | 61659088.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 107 # total simulation time in seconds |
| sim\_inst\_rate | 568029.5607 # simulation speed (in insts/sec) |
| instruction\_bus\_activity | 553255539 # total number of bit flip in instruction bus |
| instruction\_bus\_weight | 687294588 # total number of 1 count in instruction bus |
| regbank\_activity | 631135915 # total number of bit flip in register bank |
| ld\_text\_base | 0x00100000 # program text (code) segment base |
| ld\_text\_bound | 0x0017c8f8 # program text (code) segment bound |
| ld\_text\_size | 510200 # program text (code) size in bytes |
| ld\_data\_base | 0x0010b864 # program initialized data segment base |
| ld\_data\_bound | 0x002009e0 # program initialized data segment bound |
| ld\_data\_size | 1003900 # program init'ed `.data' and uninit'ed `.bss' size in bytes |
| ld\_stack\_base | 0xc0000000 # program stack segment base (highest address in stack) |
| ld\_stack\_size | 16384 # program initial stack size |
| ld\_prog\_entry | 0x00100040 # program entry point (initial PC) |
| ld\_environ\_base | 0xbfffc000 # program environment base address address |
| ld\_target\_big\_endian | 0 # target executable endian-ness, non-zero if big endian |
| mem.page\_count | 130 # total number of pages allocated |
| mem.page\_mem | 520k # total size of memory pages allocated |
| mem.ptab\_misses | 262274 # total first level page table misses |
| mem.ptab\_accesses | 183140950 # total page table accesses |
| mem.ptab\_miss\_rate | 0.0014 # first level page table miss rate |

1. ارزيابي نتایج شبیه سازی
   1. مقدمه

در این فصل به مقایسه­ی نتایج حاصل از شبیه­سازی پرداخته می­شود. کاری که انجام می­شود مقایسه ای میان انرژی مصرفی کل و زمان شبیه­سازی به ازای تغییر اندازه­ی ورودی الگوریتم خواهد بود که در نتیجه­ی آن می­توان تغییرات توان مصرفی برنامه­های بسته­ی محک را به ازای تغییر در اندازه­ی ورودیشان تحلیل کرد. همچنین می­توان برای الگوریتم­هایی که ورودی یکسانی دارند نیز مقایسه­ای بین توان مصرفیشان داشت.

* 1. ارزیابی نتایج شبیه سازی الگوریتم­های مختلف
     1. دیکسترا

1. برای ورودی آرایه­ای با ابعاد 5050:

|  |  |
| --- | --- |
| sim\_num\_insn | 121265791 # total number of instructions executed |
| sim\_total\_energy | 134447904.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 141 # total simulation time in seconds |

1. برای ورودی آرایه­ای با ابعاد 100×100:

|  |  |
| --- | --- |
| sim\_num\_insn | 1726339312 # total number of instructions executed |
| sim\_total\_energy | 135834336.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 2254 # total simulation time in seconds |

* نتیجه

با 4 برابر شدن سایز ورودی، تعداد دستورالعمل­ها تقریبا 14 برابر شده است. انرژی مصرفی 1،01 برابر شده و زمان شبیه­سازی نیز 15،98 برابر شده است. بنابراین توان مصرفی در این الگوریتم با دو برابر شدن ورودی 0،06 برابر شده است. این بدین معنی است که با افزایش سایز ورودی توان مصرفی به میزان قابل توجهی کم می­شود.

* + 1. Stringsearch

1. برای ورودی کوچک با سایز تقریبی 60 رشته:

|  |  |
| --- | --- |
| sim\_num\_insn | 445431 # total number of instructions executed |
| sim\_total\_energy | 856388.8125 # total energy consumption (nJ) |
| sim\_elapsed\_time | 1. # total simulation time in seconds |

1. برای ورودی بزرگ با سایز تقریبی 1300 رشته:

|  |  |
| --- | --- |
| sim\_num\_insn | 4473903 # total number of instructions executed |
| sim\_total\_energy | 12161095.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 9 # total simulation time in seconds |

* نتیجه

با تقریبا 22 برابر شدن سایز ورودی، تعداد دستورالعمل­ها تقریبا 10 برابر شده است. انرژی مصرفی 14.2 برابر شده و زمان شبیه­سازی نیز 9 برابر شده است. بنابراین توان مصرفی در این الگوریتم با دو برابر شدن ورودی 1،58 برابر شده است. این بدین معنی است که با افزایش سایز ورودی توان مصرفی به میزان قابل توجهی زیاد می­شود.

* + 1. بلوفیش

1. برای ورودی کوچک با سایز تقریبی 230000 کاراکتر:

|  |  |
| --- | --- |
| sim\_num\_insn | 1719563275 # total number of instructions executed |
| sim\_total\_energy | 134217744.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 2871 # total simulation time in seconds |

1. برای ورودی بزرگ با سایز تقریبی 460000 کاراکتر:

|  |  |
| --- | --- |
| sim\_num\_insn | 3701069514 # total number of instructions executed |
| sim\_total\_energy | 134217744.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 5643 # total simulation time in seconds |

* نتیجه

با دو برابر شدن سایز ورودی، با اینکه تعداد دستورالعمل­های اجرا شده تقریبا دو برابر شده است، با این حال مقدار انرژی مصرفی هیچ تغییری نکرده است. این به این معنی است که در این الگوریتم انرژی مصرفی به سایز ورودی وابستگی چندانی ندارد. با توجه به دو برابر شدن زمان به ازای دو برابر شدن سایز ورودی می­توان گفت توان مصرفی با دو برابر شدن سایز ورودی نصف می­شود.

* + 1. Rijndael

1. برای ورودی کوچک با سایز تقریبی 230000 کاراکتر:

|  |  |
| --- | --- |
| sim\_num\_insn | 473197189 # total number of instructions executed |
| sim\_total\_energy | 134521840.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 1. total simulation time in seconds |

1. برای ورودی بزرگ با سایز تقریبی 460000 کاراکتر:

|  |  |
| --- | --- |
| sim\_num\_insn | 944499149 # total number of instructions executed |
| sim\_total\_energy | 138249376.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 691 # total simulation time in seconds |

* نتیجه

با دو برابر شدن سایز ورودی، تعداد دستورالعمل­ها تقریبا دو برابر شده است. انرژی مصرفی 1.03 برابر شده و زمان شبیه­سازی نیز 1.35 برابر شده است. بنابراین توان مصرفی در این الگوریتم با دو برابر شدن ورودی 0،76 برابر شده است. این بدین معنی است که با افزایش سایز ورودی توان مصرفی به مرور کم­تر می­شود.

* + 1. Sha

1. برای ورودی کوچک با سایز تقریبی 230000 کاراکتر:

|  |  |
| --- | --- |
| sim\_num\_insn | 31054210 # total number of instructions executed |
| sim\_total\_energy | 43340804.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 64 # total simulation time in seconds |

1. برای ورودی بزرگ با سایز تقریبی 460000 کاراکتر:

|  |  |
| --- | --- |
| sim\_num\_insn | 60779163 # total number of instructions executed |
| sim\_total\_energy | 61659088.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 107 # total simulation time in seconds |

* نتیجه

با دو برابر شدن سایز ورودی، تعداد دستورالعمل­ها تقریبا دو برابر شده است. انرژی مصرفی 1،4 برابر شده و زمان شبیه­سازی نیز 1،67 برابر شده است. بنابراین توان مصرفی در این الگوریتم با دو برابر شدن ورودی 0،85 برابر شده است. این بدین معنی است که با افزایش سایز ورودی توان مصرفی به مرور کم­تر می­شود.

* + 1. مقایسه­ی الگوریتم­های بلوفیش و Rijndael

1. برای ورودی کوچک با سایز تقریبی 230000 کاراکتر:

بلوفیش:

|  |  |
| --- | --- |
| sim\_total\_energy | 134217744.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 2871 # total simulation time in seconds |

Rijndael:

|  |  |
| --- | --- |
| sim\_total\_energy | 134521840.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 513 #total simulation time in seconds |

* نتیجه

انرژی مصرفی کل به ازای ورودی کوچک برای الگوریتم بلوفیش و Rijndael با تقریب خوبی یکسان است، منتها زمان اجرای الگوریتم بلوفیش 5،6 برابر Rijndael است. لذا توان مصرفی کل برای الگوریتم Rijndael، 5،6 برابر توان مصرفی الگوریتم بلوفیش به ازای ورودی یکسان است.

1. برای ورودی بزرگ با سایز تقریبی 460000 کاراکتر:

بلوفیش:

|  |  |
| --- | --- |
| sim\_total\_energy | 134217744.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 5643 # total simulation time in seconds |

Rijndael:

|  |  |
| --- | --- |
| sim\_total\_energy | 138249376.0000 # total energy consumption (nJ) |
| sim\_elapsed\_time | 691 # total simulation time in seconds |

* نتیجه

انرژی مصرفی کل به ازای ورودی کوچک برای الگوریتم Rijndael، 1،03 برابر الگوریتم بلوفیش است، و زمان اجرای الگوریتم بلوفیش 8،17 برابر Rijndael است. لذا توان مصرفی کل برای الگوریتم Rijndael، 8،41 برابر توان مصرفی الگوریتم بلوفیش به ازای ورودی یکسان است.

1. نتيجه‌گيری و کارهای آينده
   1. نتيجه‌گيری

در بخش مطالعاتی تحقیق به این نتیجه رسیدیم که Mibench در هنگام تحلیل استاتیک و دینامیک خصوصیات عملکرد پردازنده­های تعبیه شده، خصوصیات متفاوت قابل توجهی نسبت به محک­های SPEC2000 دارد. دستورالعمل­های دینامیک تغییر بیشتری در تعداد انشعاب­ها، حافظه و عملگرهای عدد صحیح ALU دارند. همچنین سایزهای بخش داده و متن متنوعی دارد. بنابراین به عنوان جایگزین خوبی برای محک­های SPEC2000 برای تحلیل استاتیک و دینامیک خصوصیات عملکرد پردازنده­های تعبیه شده، استفاده می­شوند.

در بخش عملی پروژه هم بخشی از برنامه­های محک Mibench را مورد مطالعه قرار دادیم تا توان مصرفی این برنامه­ها را در ازای پیاده­سازی آن­ها بر روی میکروکنترلر AT91SAM7X256 به دست آوریم، و لذا با ایجاد تعییراتی در نحوه­ی ورودی گرفتن برنامه­ها آن­ها را برای شبیه­سازی آماده کردیم و با استفاده از شبیه­ساز MEET توان مصرفی و سایر مشخصه­های این برنامه­ها را به دست آوردیم. در اکثر برنامه­های بسته­ی محک Mibench، دو کد، یکی برای ورودی­های کوچک و دیگری بزرگ وجود داشت، به همین دلیل توانستیم برای توان مصرفی هر یک از این برنامه­ها در سایزهای مختلف مقایسه انجام دهیم.

* 1. کارهاي آينده

در خصوص کارهایی که که می­توان در ادامه و با استفاده از نتایچ به دست آمده انجام داد، نیز می­توان به تشخیص ناهنجاری، کشف تروجان های سخت افزاری و کشف نفوذ به سیستم از طریق مصرف توان غیر نرمال اشاره کرد.

مراجع

|  |  |
| --- | --- |
| [1] | Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, Richard B. Brown, "MiBench: A free, commercially representative embedded benchmark suite" 1301 Beal Ave., Ann Arbor, MI 48109-2122. |
| [2] | The Embedded Microprocessor Benchmark Consortium. [Online]. <http://www.eembc.com> |
| [3] | MEET Simulator version 1.1 . An step by step Guide. [Online]. <http://esrlab.ce.sharif.ir> |
| [4] | Doug Burger, " SimpleScalar Tutorial " Computer Sciences Department University of Wisconsin-Madison. |

واژه نامه

بخش الف: واژه نامه فارسي به انگليسي

Simulation Statistics آمار شبیه­سازی

Office Automation اتوماسیون اداری

Automotive and Industrial Control اتوماسیون و کنترل صنعتی

Telecommunications ارتباطات

Digital Signature امضای دیجیتال

Security امنیت

Energy Consumption انرژی مصرفی

Branch انشعاب

Recursive بازگشتی

platforme بستر نرم­افزاری

Immediate بلافصل

Optimized بهینه شده

Estimation تخمین

Power Consumption توان مصرفی

Pipeline خط لوله

Manipulation دستکاری

Encrypt رمزگذاری

Decrypt رمزگشایی

Embeded system سیستم تعبیه­شده

Desktop system سیستم رومیزی

Host system سیستم میزبان

Networking شبکه

Simulation شبیه­سازی

Associative شرکت پذیری

Nonrecursive غیر بازگشتی

Portable قابل حمل

Consumer Devices قطعات مصرف کننده

Performance کارایی

General-purpose computers کامپیوترهای همه منظوره

Symmetric متقارن

Benchmark محک

Validation معتبرسازی

Floating point ممیز شناور

Parallelism موازی­سازی

Asymmetric نامتقارن

Nibble نیم­بایت

Defragmentation یکپارچه­سازی

بخش ب: واژه نامه انگليسي به فارسي

Asymmetric نامتقارن

Associative شرکت پذیری

Automotive and Industrial Control اتوماسیون و کنترل صنعتی

Benchmark محک

Branch انشعاب

Consumer Devices قطعات مصرف کننده

Decrypt رمزگشایی

Defragmentation یکپارچه­سازی

Desktop system سیستم رومیزی

Digital Signature امضای دیجیتال

Embeded system سیستم تعبیه­شده

Encrypt رمزگذاری

Energy Consumption انرژی مصرفی

Estimation تخمین

Floating point ممیز شناور

General-purpose computers کامپیوترهای همه منظوره

Host system سیستم میزبان

Immediate بلافصل

Manipulation دستکاری

Networking شبکه

Nibble نیم­بایت

Nonrecursive غیر بازگشتی

Optimized بهینه شده

Simulation Statistics آمار شبیه­سازی

Office Automation اتوماسیون اداری

Parallelism موازی­سازی

Performance کارایی

Pipeline خط لوله

Platforme بستر نرم­افزاری

Portable قابل حمل

Power Consumption توان مصرفی

Recursive بازگشتی

Security امنیت

Simulation شبیه­سازی

Symmetric متقارن

Telecommunications ارتباطات

Validation معتبرسازی

**Abstract**

One of the most important parameters in designing embedded systems, is their power consumption. It’s because of two reasons: 1) These systems use batteries to work. 2) Weight, volume and cost of them is very important to be minimum. Therefore it’s not possible to use common heat sinks of digital systems in them. The objective of this project is all of programs of Mibench suite. Mibench is a standard suite for embedded applications. In this project, programs of Mibench suite have been executed on ARM-based embedded system, and their power consumption have been extracted and analysed for different parts of program.

**Keywords:** Embeded systems, Benchmark, Mibench



Iran University of Science and Technology

School of Computer Engineering

Power consumption evaluation and analysis of different programs of Mibench benchmark upon an embedded system based on ARM proccessors

A Thesis Submitted in Partial Fulfillment of the Requirement for the Degree of Master of Science in Computer Engineering - Hardware

By:

Zeinab Mahdavi

Supervisor:

Dr. Mehdi Fazeli

November 2013

1. Floating point [↑](#footnote-ref-1)
2. Standard Performance Evaluation Corporation [↑](#footnote-ref-2)
3. General-perpose [↑](#footnote-ref-3)
4. Embeded [↑](#footnote-ref-4)
5. EDN Embedded Microprocessor BenchmarkConsortium [↑](#footnote-ref-5)
6. Automotive and Industrial Control [↑](#footnote-ref-6)
7. Consumer Devices [↑](#footnote-ref-7)
8. Office Automation [↑](#footnote-ref-8)
9. Networking [↑](#footnote-ref-9)
10. Security [↑](#footnote-ref-10)
11. Telecommunications [↑](#footnote-ref-11)
12. Platforme [↑](#footnote-ref-12)
13. Automotive and Industrial Control [↑](#footnote-ref-13)
14. Basicmath [↑](#footnote-ref-14)
15. Bitcount [↑](#footnote-ref-15)
16. Qsort [↑](#footnote-ref-16)
17. Susan [↑](#footnote-ref-17)
18. Constant [↑](#footnote-ref-18)
19. Look-up table [↑](#footnote-ref-19)
20. Susan [↑](#footnote-ref-20)
21. threshold [↑](#footnote-ref-21)
22. network [↑](#footnote-ref-22)
23. patricia [↑](#footnote-ref-23)
24. Dijkstra [↑](#footnote-ref-24)
25. Patricia [↑](#footnote-ref-25)
26. Routing algorithm [↑](#footnote-ref-26)
27. security [↑](#footnote-ref-27)
28. hashing [↑](#footnote-ref-28)
29. Advanced Encryption Standard [↑](#footnote-ref-29)
30. Secure Hash Algorithm [↑](#footnote-ref-30)
31. [Pretty Good Privacy](https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0CDkQFjAA&url=http%3A%2F%2Fen.wikipedia.org%2Fwiki%2FPretty_Good_Privacy&ei=ffVvUouNMcW_igLw1IGYBg&usg=AFQjCNGBYKaQBhKuWBn3xit0MTrskh9-Yw&bvm=bv.55123115,d.cGE&cad=rja) [↑](#footnote-ref-31)
32. phil Zimmerman [↑](#footnote-ref-32)
33. consumer devices [↑](#footnote-ref-33)
34. Personal Digital Assistants [↑](#footnote-ref-34)
35. overhead [↑](#footnote-ref-35)
36. simplescalar [↑](#footnote-ref-36)
37. office Automation [↑](#footnote-ref-37)
38. Stringsearch [↑](#footnote-ref-38)
39. case insensitive comparision algorithm [↑](#footnote-ref-39)
40. Telecommunications [↑](#footnote-ref-40)
41. Checksum [↑](#footnote-ref-41)
42. [Fast Fourier Transform](http://acronyms.thefreedictionary.com/Inverse+Fast+Fourier+Transform) [↑](#footnote-ref-42)
43. [Inverse Fast Fourier Transform](http://acronyms.thefreedictionary.com/Inverse+Fast+Fourier+Transform) [↑](#footnote-ref-43)
44. Global Standard for Mobile [↑](#footnote-ref-44)
45. Time-Division Multiple Access [↑](#footnote-ref-45)
46. Frequency-Division Multiple Access [↑](#footnote-ref-46)
47. [Adaptive Differential Pulse-Code Modulation](http://en.wikipedia.org/wiki/Adaptive_differential_pulse-code_modulation) [↑](#footnote-ref-47)
48. [Pulse-Code Modulation](http://en.wikipedia.org/wiki/Adaptive_differential_pulse-code_modulation) (PCM) [↑](#footnote-ref-48)
49. Cyclic Redundancy Check [↑](#footnote-ref-49)
50. Rebel Netwinder [↑](#footnote-ref-50)
51. DRAM [↑](#footnote-ref-51)
52. microbenchmark [↑](#footnote-ref-52)
53. cheetah [↑](#footnote-ref-53)
54. Instruction Distribution [↑](#footnote-ref-54)
55. control [↑](#footnote-ref-55)
56. integer [↑](#footnote-ref-56)
57. Floating point [↑](#footnote-ref-57)
58. memory [↑](#footnote-ref-58)
59. Branches [↑](#footnote-ref-59)
60. Branch Target Buffer [↑](#footnote-ref-60)
61. random access patterns [↑](#footnote-ref-61)
62. Instruction Per Cycle [↑](#footnote-ref-62)
63. parallelism [↑](#footnote-ref-63)
64. defragmentation [↑](#footnote-ref-64)
65. Microcontroller Energy Estimation Tool [↑](#footnote-ref-65)
66. „An Accurate Instruction‐Level Energy Estimation Model and Tool for Embedded Systems“ [↑](#footnote-ref-66)
67. Options [↑](#footnote-ref-67)
68. Simulation statistics [↑](#footnote-ref-68)
69. Hard-Code [↑](#footnote-ref-69)
70. Optimized 1 bit/loop counter [↑](#footnote-ref-70)
71. recursive bit count by nibbles [↑](#footnote-ref-71)
72. non-recursive bit count by nibbles using a table look-up [↑](#footnote-ref-72)
73. non-recursive bit count by bytes using a table look-up [↑](#footnote-ref-73)
74. Shift and count bits [↑](#footnote-ref-74)
75. Ratko Tomic [↑](#footnote-ref-75)
76. Bob Stout [↑](#footnote-ref-76)
77. Auke Reitsma and Bruce Wedding [↑](#footnote-ref-77)
78. case insensitive comparision algorithm [↑](#footnote-ref-78)
79. Pratt-Boyer-Moore string search [↑](#footnote-ref-79)
80. Thad Smith [↑](#footnote-ref-80)
81. Joan Daemen [↑](#footnote-ref-81)
82. Vincent Rijmen [↑](#footnote-ref-82)
83. [National Institute of Standards and Technology](http://www.nist.gov/) [↑](#footnote-ref-83)