دانشکده مهندسی کامپیوتر

ارزیابی و تحلیل توان مصرفی برنامه های مختلف محک استاندارد MiBenchبر روی یک سیستم تعبیه شده مبتنی بر پردازنده های ARM

پایان‌نامه برای دریافت درجه کارشناسی

در رشته مهندسی کامپیوتر - گرایش سخت‌افزار

نام دانشجو:

زینب مهدوی

استاد راهنما:

دکتر مهدی فاضلی

آبان ماه ۱۳۹۲

دانشکده مهندسی کامپیوتر

ارزیابی و تحلیل توان مصرفی برنامه های مختلف محک استاندارد MiBenchبر روی یک سیستم تعبیه شده مبتنی بر پردازنده های ARM

پایان‌نامه برای دریافت درجه کارشناسی

در رشته مهندسی کامپیوتر - گرایش سخت‌افزار

نام دانشجو:

زینب مهدوی

استاد راهنما:

دکتر مهدی فاضلی

آبان ماه ۱۳۹۲


بسم الله الرحمن الرحيم

تأییدیه هیأت داوران جلسه دفاع از پایان‌نامه/رساله

نام دانشکده: مهندسی کامپیوتر

نام دانشجو: زینب مهدوی

عنوان پایان‌نامه: ارزیابی و تحلیل توان مصرفی برنامه‌های مختلف محک استاندارد MiBench بر روی یک سیستم تعبیه شده مبتنی بر پردازنده‌های ARM

تاریخ دفاع: آبان ماه ۱۳۹۲

رشته: مهندسی کامپیوتر

گرایش: سخت‌افزار

ردیف	سمت	نام و نام خانوادگی	مرتبه دانشگاهی	دانشگاه یا مؤسسه	امضاء
۱	استاد راهنما	مهدی فاضلی	استادیار	دانشگاه علم و صنعت ایران	
۲	استاد مدعو داخلی	سید وحید ازهری	استادیار	دانشگاه علم و صنعت ایران	

تأییدیه صحت و اصالت نتایج

بسمه تعالی

اینجانب زینب مهدوی به شماره دانشجویی ۸۸۵۲۲۲۴۳ دانشجوی رشته کامپیوتر، گرایش سخت‌افزار در مقطع تحصیلی کارشناسی ارشد تأیید می‌نمایم که کلیه‌ی نتایج این پایان‌نامه حاصل کار اینجانب و بدون هرگونه دخل و تصرف است و موارد نسخه‌برداری‌شده از آثار دیگران را با ذکر کامل مشخصات منبع ذکر کرده‌ام. درصورت اثبات خلاف مندرجات فوق، به تشخیص دانشگاه مطابق با ضوابط و مقررات حاکم (قانون حمایت از حقوق مؤلفان و مصنفان و قانون ترجمه و تکثیر کتب و نشریات و آثار صوتی، ضوابط و مقررات آموزشی، پژوهشی و انضباطی) با اینجانب رفتار خواهد شد و حق هرگونه اعتراض در خصوص احقاق حقوق مکتسب و تشخیص و تعیین تخلف و مجازات را از خویش سلب می‌نمایم. در ضمن، مسئولیت هرگونه پاسخگویی به اشخاص اعم از حقیقی و حقوقی و مراجع ذی‌صلاح (اعم از اداری و قضایی) به عهده اینجانب خواهد بود و دانشگاه هیچ‌گونه مسؤولیتی در این خصوص نخواهد داشت.

نام و نام خانوادگی: زینب مهدوی

امضا و تاریخ:

مجوز بهره‌برداری از پایان‌نامه

بهره‌برداری از این پایان‌نامه در چارچوب مقررات کتابخانه و با توجه به محدودیتی که توسط استاد راهنما به شرح زیر تعیین می‌شود، بلامانع است:

☐ بهره‌برداری از این پایان‌نامه برای همگان بلامانع است.

☐ بهره‌برداری از این پایان‌نامه با اخذ مجوز از استاد راهنما، بلامانع است.

☐ بهره‌برداری از این پایان‌نامه تا تاریخ ..... ممنوع است.

نام استاد یا اساتید راهنما:

تاریخ:

امضا:

تقدیم به ....(اختیاری)

با تشکر از ...(اختیاری)

چکیده

یکی از پارامترهای مهم در طراحی سیستم‌های تعبیه شده میـزان تـوان مصـرفی در آن‌هاسـت. ایـن مسـئله عمدتا به دو دلیل است: ۱) این سیستم‌ها برای کار از باطری استفاده می‌کنند، ۲) وزن، حجم و هزینـه‌ی آن-ها بسیار مهم است، لذا نمی‌توان از وسایل خنک‌کننده‌ی مرسوم در سیستم‌های دیجیتـال در آن‌هـا اسـتفاده نمود. در این پروژه هدف بر این است که کل برنامه‌های محک بسته MiBench که یک بسته اسـتاندارد بـرای کاربردهای تعبیه شده‌است بر روی یک سیستم تعبیه شده مبتنی بر ARM اجرا شده و میزان تـوان مصـرفی آن‌ها به ازای قسمت‌های مختلف برنامه استخراج گردیده و تحلیل شود.

واژه‌های کلیدی: سیستم‌های تعبیه شده، محک، Mibench

فهرست مطالب

عنوان **صفحه**

فهرست شکل‌ها

عنوان **صفحه**

فهرست جدول‌ها

عنوان **صفحه**

فـصل ۱ :

مقدمه

۱-۱- شرح مسأله

امروزه سیستم‌های تعبیه شده به میزان زیادی در حوزه‌های مختلف مورد استفاده قرار می‌گیرند، به طوری که بنابر برخی گزارشات، بیش از ۹۹٪ پردازنده‌های تولید شده در سیستم‌های تعبیه شده استفاده شده‌اند. تعاریف متعددی برای سیستم‌های تعبیه شده در ادبیات موضوعی وجود دارد. به طور کلی، منظور از یک سیستم تعبیه شده، سیستمی دیجیتالی است که کنترل یک وظیفه را در سیستمی بزرگ‌تر به عهده می‌گیرد و تنها انجام همین یک وظیفه 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.

یکی از پارامترهای مهم در طراحی سیستم‌های تعبیه شده میزان توان مصرفی در آن‌هاست. این مسئله عمدتا به دو دلیل است: ۱) این سیستم‌ها برای کار از باطری استفاده می‌کنند، ۲) وزن، حجم و هزینه در آن‌ها بسیار مهم است، لذا نمی‌توان از وسایل خنک‌کننده مرسوم در سیستم‌های دیجیتال در آن‌ها استفاده نمود. در این پروژه هدف بر این است که کل برنامه‌های محک بسته MiBench که یک بسته استاندارد برای کاربردهای تعبیه شده است بر روی یک سیستم تعبیه شده مبتنی بر ARM اجرا شده و میزان توان مصرفی آن‌ها به ازای قسمت‌های مختلف برنامه استخراج گردیده و تحلیل شود. با استفاده از این تحلیل می‌توان یک راهنما برای نحوه‌ی توسعه برنامه‌ی این سیستم‌ها ارائه داد.

فـصل ۲ :

تعاریف و مفاهیم مبنایی

۲-۱- 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[۱] 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[۲] 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[۳] 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[۴] 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

---

۱ Floating point

۲ Standard Performance Evaluation Corporation

۳ General-perpose

۴ Embeded

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:

۱) پروسسورهای تعبیه شده معمولا برای اینکه به طور همزمان در پروسه طراحی به حساب بیایند، نیاز به توان دارند.

۲) 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 EEMBC[۱] 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 ۳۵ 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 ۶ 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[۲]، قطعات مصرف کننده[۳]، اتوماسیون اداری[۴]، شبکه[۵]، امنیت[۶] و ارتباطات[۷]. 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

---

۱ EDN Embedded Microprocessor BenchmarkConsortium

۲ Automotive and Industrial Control

۳ Consumer Devices

۴ Office Automation

۵ Networking

۶ Security

۷ Telecommunications

انتقال به هر بستر نرم افزاری[۱]ای که پشتیبانی کامپایلر را دارد، می‌باشد.

در ادامهٔ این بخش می‌خوانیم که بخش ۲-۲ محک‌ها و مجموعه داده‌ها در Mibench را توصیف می‌کند. در بخش ۲-۳ 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 ۲-۴ 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 ۲-۵ خلاصه‌ای از مشخصات حجم کاری تعبیه شده موجود در آزمایش‌های ما تهیه شده‌اند.[1]

۲-۲- 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.

---

۱ Platforme

جدول (۲-۱) محک‌های Mibench

Auto./Industrial	Consumer	Office	Network	Security	Telecomm.
basicmath	jpeg	ghostscript	dijkstra	blowfish enc.	CRC32
bitcount	lame	ispell	patricia	blowfish dec.	FFT
qsort	mad	rsynth	(CRC32)	pgp sign	IFFT
susan (edges)	tiff2bw	sphinx	(sha)	pgp verify	ADPCM enc.
susan (corners)	tiff2rgba	stringsearch	(blowfish)	rijndael enc.	ADPCM dec.
susan (smoothing)	tiffdither			rijndael dec.	GSM enc.
	tiffmedian			sha	GSM dec.
	typeset				

۲-۲-۱- اتوماسیون و کنترل صنعتی[۱]

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[۲]، آزمون شمارش بیتی[۳]، یک الگوریتم مرتب سازی[۴]، و یک برنامه ی تشخیص شکل[۵] 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[۶]ها می‌باشد.

۱ Automotive and Industrial Control
۲ Basicmath
۳ Bitcount
۴ Qsort
۵ Susan
۶ Constant

◘ 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 ۴ بیتی، شمارش غیر بازگشتی ۴ بیتی با استفاده از جدول جستجو[۱]، شمارش غیر بازگشتی ۸ بیتی با استفاده از جدول جستجو و شیفت دادن و شمارش بیت‌ها. در اینجا داده‌ی ورودی، یک آرایه‌ی صحیح از اعداد صحیح با مقادیر '۰' و '۱' 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[۲]

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 چشم-انداز مبتنی بر برنامه‌ی تضمین کیفیت به کار گرفته می‌شود. سوزان همچنین تصویر را صاف می‌کند و تنظیمات آن مانند آستانه[۳]، 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.

۲-۲-۲- شبکه[۴]

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

---

۱ Look-up table
۲ Susan
۳ threshold
۴ network

کردن کوتاه‌ترین مسیر در یک گراف و ایجاد و جستجو کردن درخت ساختمان داده‌ی پاتریشیا[۱] 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[۲]

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 $O(n^2)$ اجرا می‌شود.

◘ پاتریشیا[۳]

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[۴] مورد استفاده قرار می‌گیرد. داده‌ی ورودی برای این محک، یک لیست از ترافیک IP از یک سرور بسیار فعال برای یک دوره‌ی ۲ ساعته است. شماره‌های IP شماره‌های تبدیل شده هستند.

۲-۲-۳- امنیت[۵]

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[۶] داده‌ها می‌باشد. یکی از این الگوریتم‌ها rijndael است که یک استاندارد رمزگذاری جدید[۷] (AES) است. سایر الگوریتم‌های امنیتی نیز بلوفیش، پی‌جی‌پی و SHA هستند.

---

۱ patricia
۲ Dijkstra
۳ Patricia
۴ Routing algorithm
۵ security
۶ hashing
۷ Advanced Encryption Standard

◘ الگوریتم رمزگذاری/ رمزگشایی بلوفیش

یک نوع رمزنگاری متقارن بلوک با یک کلید (key) با طول متغیر است. این الگوریتم در سال ۱۹۹۲ توسط بروس اشنایدر توسعه داده شد. از آن‌جا که طول کلید آن می‌تواند بین ۳۲ بیت تا ۴۴۸ 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[۱] sha

یک الگوریتم امن درهم‌سازی است که یک پیام خلاصه‌ی ۱۶۰ 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 ۱۲۸، ۱۹۲ و ۲۵۶ بیتی از کلید و بلوک‌هاست. مجموعه داده‌های ورودی آن همانند بلوفیش است.

◘ الگوریتم رمزگذاری/ رمزگشایی پی‌جی‌پی[۲]

یک الگوریتم عمومی رمزنگاری کلید است که توسط فیل زیمرمن[۳] 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.

---

۱ Secure Hash Algorithm

۲ Pretty Good Privacy

۳ phil Zimmerman

۲-۲-۴- دستگاه‌های مصرف کننده[۱]

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[۲] (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

---

۱ consumer devices

۲ Personal Digital Assistants

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 ۱، لایه ۲ و لایه ۳ 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[۱] 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[۲] است.

۲-۲-۵- اتوماسیون اداری[۳]

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 آن-ها 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[۴]

این محک با استفاده از الگوریتم مقایسه‌ی حساس به حروف[۵] در عبارات به دنبال کلمات داده شده می‌گردد.

---

۱ overhead

۲ simplescalar

۳ office Automation

۴ Stringsearch

۵ case insensitive comparision algorithm

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.

۲-۲-۶- ارتباطات[۱]

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[۲] می‌باشد.

تبدیل سریع فوریه[۳]/ معکوس تبدیل سریع فوریه[۴]

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.

---

۱ Telecommunications

۲ Checksum

۳ Fast Fourier Transform

۴ Inverse Fast Fourier Transform

◘ رمزگذاری/ رمزگشایی جی‌اس‌ام[۱]

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[۲] یا FDMA[۳] 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[۴]

مدولاسیون کد پالس دیفرانسیل تطابقی یک نوع از استاندارد شناخته شده‌ی مدولاسیون کد پالس[۵] است. یک اجرای متداول از این محک این است که یک نمونه مدولاسیون کد پالس خطی ۱۶ بیتی را می‌گیرد و به یک نمونه ۴ بیتی تبدیل می‌کند. نرخ فشرده سازی در اینجا ۴ به ۱ است. داده‌های ورودی در اینجا نمونه بزرگ و کوچک از گفتار است.

◘ کد افزونگی دوره ای ۳۲ بیتی[۶] (CRC32)

این محک یک عملیات ۳۲ 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.

۲-۳- معتبرسازی مدل میکرومعماری

پیکربندی کنونی در جدول ۳ ، 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

---

۱ Global Standard for Mobile

۲ Time-Division Multiple Access

۳ Frequency-Division Multiple Access

۴ Adaptive Differential Pulse-Code Modulation

۵ Pulse-Code Modulation (PCM)

۶ Cyclic Redundancy Check

برابر ایستگاه کاری توسعه دهنده‌ی ریل نت‌ویندر[۱] معتبر سازی کرده‌ایم. نت‌ویندر شامل یک ریزپردازنده strongARM SA-110 با فرکانس ۲۷۵ مگاهرتز، ۱۲۸ مگابایت حافظه‌ی دینامیک[۲] 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[۳]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 ۳,۲ 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.

۲-۴- 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 محک-های 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 ۲-۲ نمایش داده شده، مجموعه‌ی داده‌های کوچک برای Mibench تقریبا ۵۰ میلیون دستورالعمل دینامیک و برای مجموعه داده‌های بزرگ بیش از ۷۵۰ میلیون دستورالعمل دینامیک است. داده‌های عملکرد کش به وسیله‌ی شبیه‌سازی مراجع حافظه‌ی همه‌ی محک‌ها، با استفاده از چیتا[۴] 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.

---

۱ Rebel Netwinder
۲ DRAM
۳ microbenchmark
۴ cheetah

جدول (۲-۲) سایز دستورالعمل‌های محک‌های Mibench

Benchmark	Small Instruction Count	Large Instruction Count	Benchmark	Small Instruction Count	Large Instruction Count
basicmath	65,459,080	1,000,000,000	ispell	8,378,832	640,420,106
bitcount	49,671,043	384,803,644	rsynth	57,872,434	85,005,687
qsort	43,604,903	595,400,120	stringsearch	158,646	38,960,051
susan.corners	1,062,891	586,076,156	blowfish.decode	52,400,008	737,920,623
susan.edges	1,836,965	732,517,639	blowfish.encode	42,407,674	246,770,499
susan.smoothing	24,897,492	1,000,000,000	pgp.decode	85,006,293	259,293,845
jpeg.decode	6,677,595	990,912,065	pgp.encode	38,960,650	824,946,344
jpeg.encode	28,108,471	543,976,667	rijndael.decode	23,706,832	140,889,705
lame	175,190,457	544,057,733	rijndael.encode	3,679,378	24,910,267
mad	25,501,771	272,657,564	sha	13,541,298	20,652,916
tiff2bw	34,003,565	697,493,266	CRC32	52,839,894	61,659,073
tiff2rgba	36,948,939	1,000,000,000	FFT.inverse	65,667,015	377,253,252
tiffdither	273,926,642	1,000,000,000	FFT	52,625,918	143,263,412
tiffmedian	141,333,005	817,729,663	adpcm.decode	30,159,188	151,699,690
typeset	23,395,912	84,170,256	adpcm.encode	37,692,050	832,956,169
dijkstra	64,927,863	272,657,564	gsm.decode	23,868,371	548,023,092
patricia	103,923,656	1,000,000,000	gsm.encode	55,361,308	472,171,446
ghostscript	286,770,117	673,391,179			

۲-۴-۱- توزیع دستورالعمل‌ها[۱]

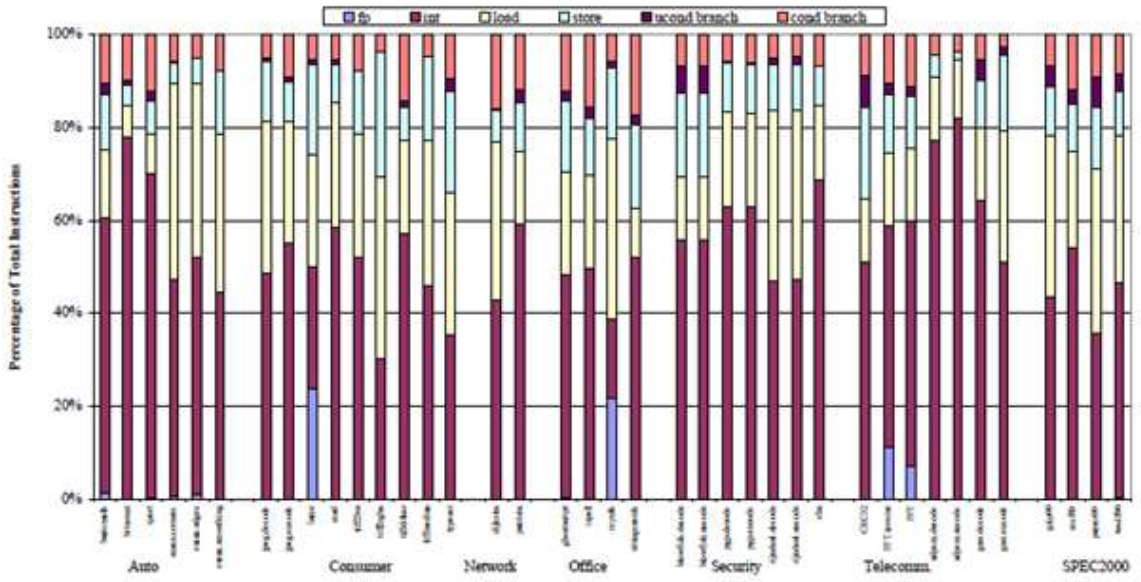
چهار کلاس اصلی از دستورالعمل‌ها وجود دارد:

- کنترل[۲] (انشعابات شرطی و غیر شرطی)
- عدد صحیح[۳]
- ممیز شناور[۴]
- حافظه[۵] (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

۱ Instruction Distribution
۲ control
۳ integer
۴ Floating point
۵ memory

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 ۲-۱ نحوه‌ی توزیع تمام برنامه‌های Mibench و SPEC2000 را نشان می‌دهد.



شکل (۲-۱) 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 ۵۰ 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/رمزگشایی ADPCM در مقایسه با هرکدام از محک‌های SPEC که ماکزیمم ۵۰ درصد از دستورات عدد صحیح ALU را دارند، حدوداً ۸۰ 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 شده-ی مختلف هستند. بررسی روی مجموعه‌ی کل محک‌ها نشان می‌دهد که تنوع، وقتی که آن‌ها را به صورت

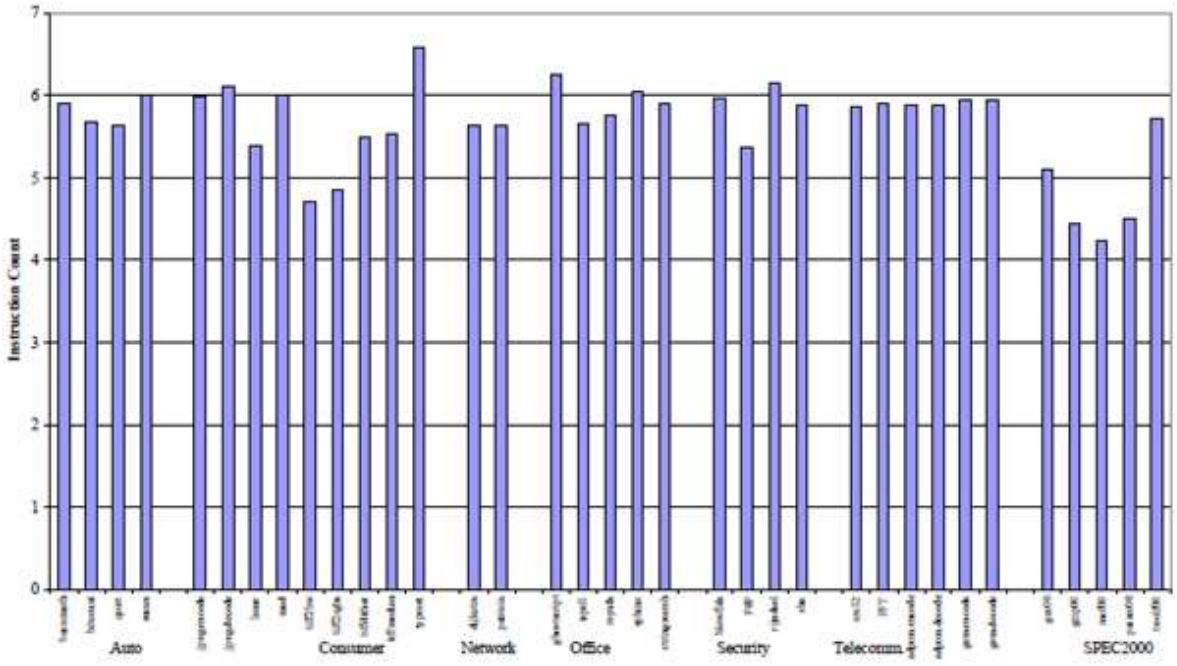
کلی در نظر می‌گیریم بیشتر است. به عنوان مثال، تعداد انشعابات در Mibench تنوع بسیار کمی دارد. آزمون شمارش بیتی (bitcount)، رمزگذار ADPCM و همچنین چندین محک دیگر، کمتر از ۱۰ درصد از عملیات انشعابی برای برنامه‌های محاسبات فشرده استفاده می‌کنند. محک‌هایی مانند بلوفیش، tiff2rgba و tiffmedian، کمتر از ۶ درصد دستورات انشعاب دارند. بیشترین انشعابات متعلق به محک‌های متنی است. Stringsearch، ispell و CRC32 در محک ارتباطات در رنجی بین ۱۸ تا ۲۰ درصد از دستورات انشعاب استفاده می‌کنند. محک‌های SPEC، به جز gzip00 که فقط ۹ درصد دستورات انشعاب دارد، معمولاً بیشتر از ۱۵ 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 ۵۰ 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 ۴۰ 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.

۲ – ۴ – ۲ – انشعابات[۱]

Mibench در تعداد انشعابات کاملاً متنوع است. تعداد انشعابات در برخی از محک‌ها کم است، به شکل ۲-۲ 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 ۶ دارد و تقریباً بقیه بالای 5.5 هستند. تعداد کمی از محک‌های مصرف‌کننده هم هستند که همانند SPECها سایز زیر ۵ دارند.

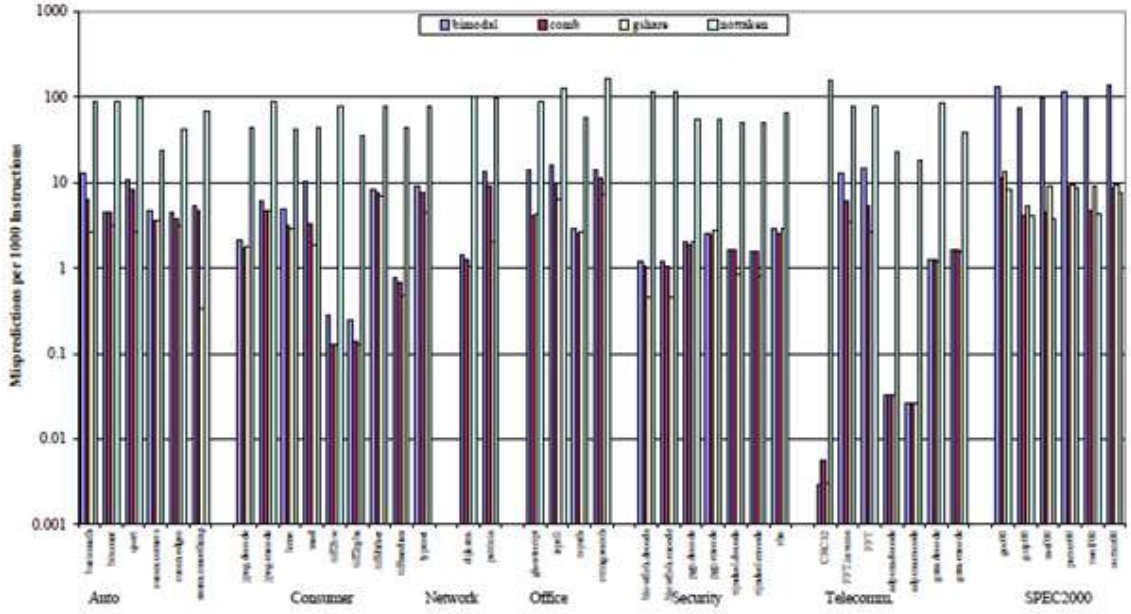
۱ Branches



شکل (۲-۲) 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 ۲-۳ نشان داده شده است. همه‌ی پیش‌بینی کننده‌ها با استثنای استراتژی not-taken از یک 8k BTB[۱] 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.

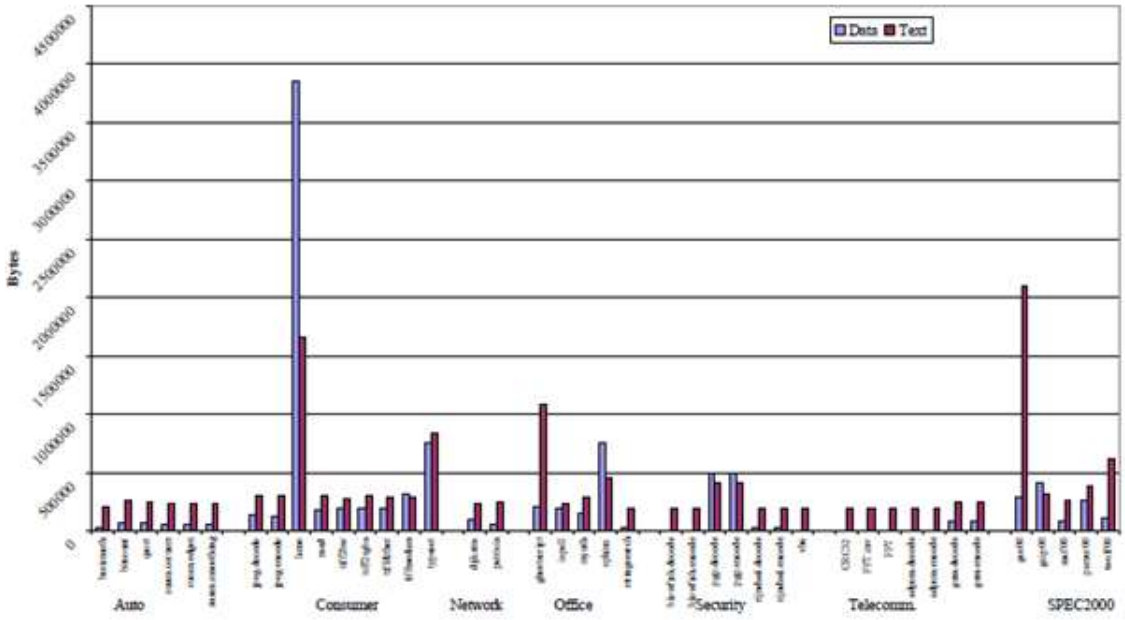
۱ Branch Target Buffer



شکل (۲-۳) نرخ پیش‌بینی انشعاب برای محک‌های Mibench و SPEC2000

۲-۴-۳- حافظه

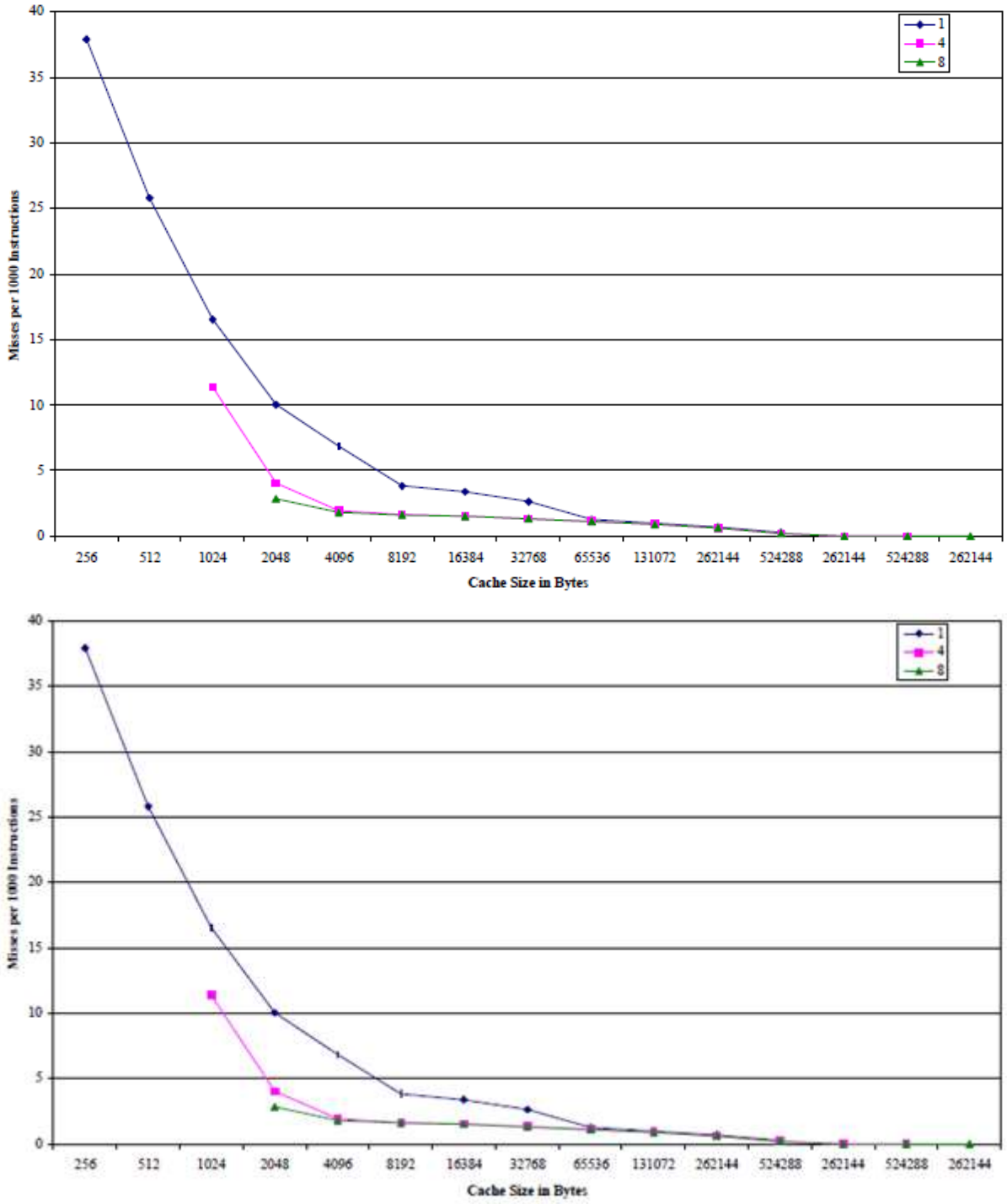
در کنار توزیع دستورالعمل‌ها و قابلیت پیش‌بینی انشعابات، رفتار حافظه یکی دیگر از نکات مهم قابل ملاحظه به هنگام ارزیابی حجم کار تعبیه شده است. سایز ایستای حافظه و قابلیت کش کردن حافظه با SPEC مقایسه شده است. در شکل ۲-۴، سایز متن‌ها و داده‌های Mibench و SPEC2000 نشان داده شده است. این دو مجموعه محک سایز 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 Mb 2 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 Mb 0.5دارد (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.



شکل (۲-۴) 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 ۱۷۵ تا ۲۰۰ 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.

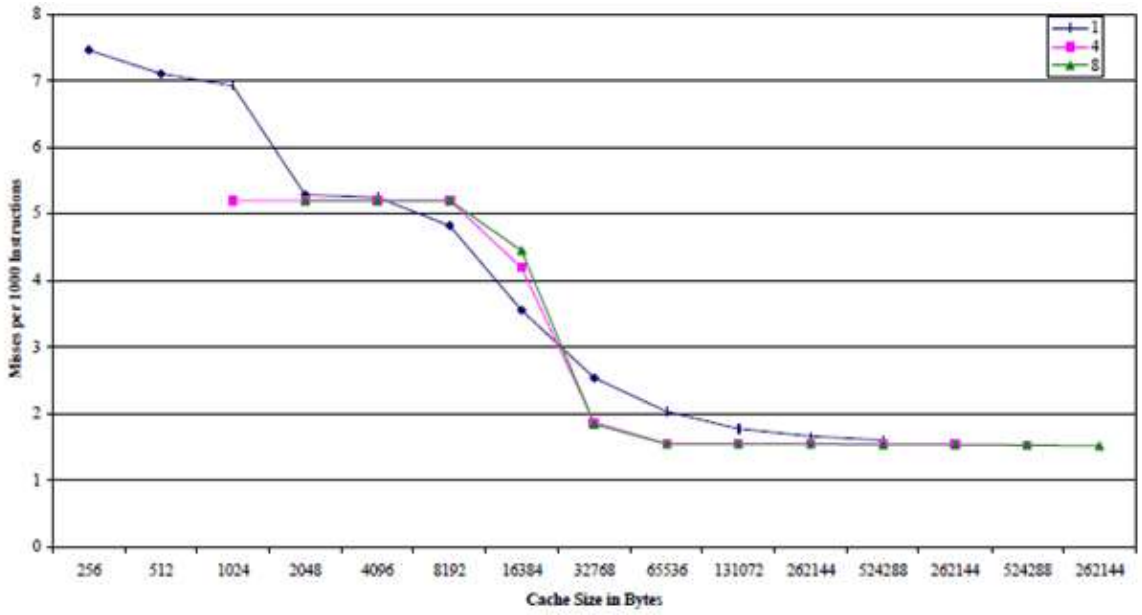
همان‌طور که در بالا نشان داده شده است، محک‌ها در Mibench سایزهای متفاوتی برای مجموعه‌ی داده‌ها دارند. به این دلیل، Mibench بر روی برخی از محک‌ها نرخ خطای کش مشابه و برای بقیه‌ی آن‌ها نرخ کم‌تری دارد.شکل ۲-۵ و ۲-۶، نرخ خطای کش را برای تعداد مجموعه‌های متفاوت برای برخی از محک‌ها در Mibench نشان داده است. 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 ۲-۵ می‌باشند. بیشتر شکل‌های نرخ خطا (برای محک‌های مختلف)، با ازای شرکت-پذیری بیشتر از 4-way و سایز بزرگ‌تر از ۸ کیلوبایت، نرخ خطای ناچیزی دارند.



شکل (۲-۵) نرخ خطای کش برای الگوریتم‌های Rijndael (شکل بالا) و Ispell (شکل پایین) با خطوط ۱۶ بیتی

همچنین با توجه به شکل ۲-۵، برای اغلب محک‌های Mibench، نرخ خطا تا حدود کمتر از ۲ درصد در محدوده‌ی ۴ تا ۶ کیلوبایت به طور زیادی افت می‌کنند. برخی از محک‌های SPEC2000 نرخ‌های خطا را تا محدوده‌ی ۶ تا ۳۲ کیلوبایت، زیر ۲ درصد کاهش نمی‌دهد که این مقدار نسبتاً بزرگی است. بعضی از محک-

ها مثل gcc00 و patricia به واسطه‌ی الگوهای دستیابی تصادفی[۱]، نیاز به شرکت‌پذیری بیشتری دارند. اما در واقع 8-way برای پایین آوردن نرخ خطا کافی است.



شکل (۲-۶) نرخ خطای کش برای الگوریتم tiff2rgba با خطوط ۱۶ 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 ۲۵۶ یا ۵۱۲ برای مجموعه‌ها، همان‌طور که قبلاً توضیح داده شد کافی است.

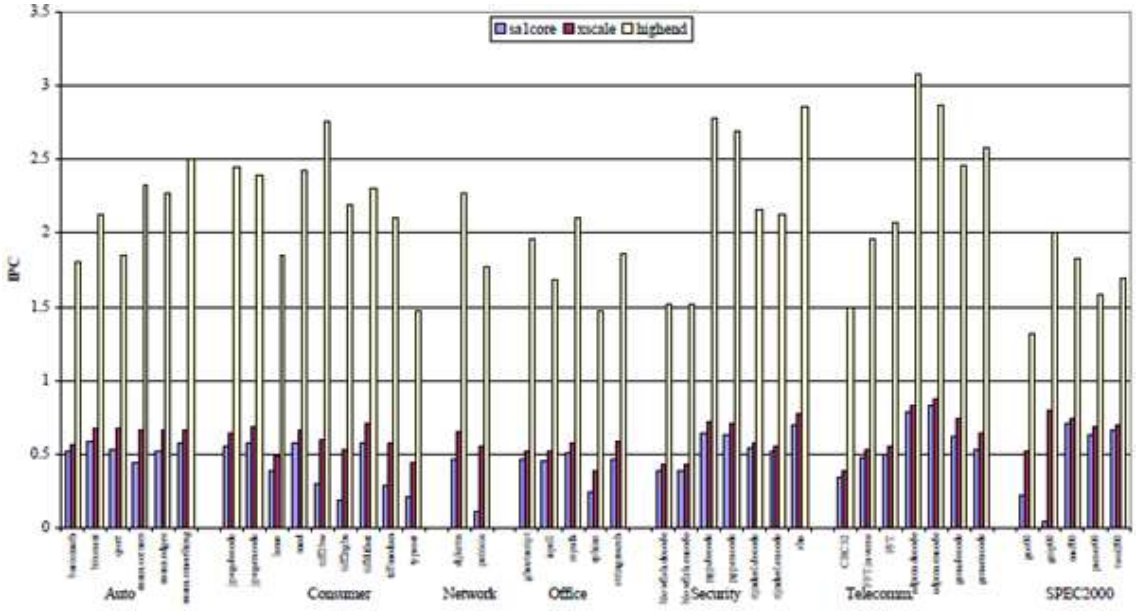
۲-۴-۴- عملکرد محک

برای انجام یک تحلیل IPC[۲] در Mibench، ۳ میکروساختار مختلف با simplescalar/ARM شبیه‌سازی شده‌اند. ساختار این ماشین‌ها در جدول ۲-۳ 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.

۱ random access patterns

۲ Instruction Per Cycle

نمایش شبیه‌سازی‌ها با هر یک از این ساختارها، در شکل ۲-۷ 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 ۲ تا ۳ برابر IPC ساختارهای نسل کنونی و آینده دست می‌یابد.



شکل (۲-۷) 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[۱] به وسیله‌ی طرح not-taken رنج می‌برد.. این مسئله به طور نامحسوسی در شکل ۲-۷ 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 پایه‌ی بزرگ و انشعابات قابل پیش‌بینی ساده‌ای دارند، احتمالاً منابع

۱ parallelism

کافی برای انجام همه‌ی دستورالعمل‌های موازی وجود ندارد.

جدول (۲-۳) تنظیمات ARM

	Current	Next Generation	High-end
Fetch queue (instructions)	2	4	32
Branch Predictor	Not-taken	8k Bimodal, 2k 4-way BTB	Combining: 4k Bimodal, 4k Gshare, 1k 4-way BTB
Fetch & Decode width	1	1	4
Issue width	1 (In-order)	1 (In-order)	4 (Out-of-order)
Functional units	1 int ALU, 1 FP mult, 1 FP ALU	1 int ALU, 1 FP mult, 1 FP ALU	1 int ALU, 1 FP mult, 4 FP ALU
Instruction L1 Cache	16 k, 32-way	32 k, 32-way	64 k, 2-way
Data L1 Cache	16 k, 32-way	32 k, 32-way	64 k, 2-way
L2 Cache	None	None	512 k, 4-way, unified
Memory (bus width, first block latency)	4-byte, 12 cycle	4-byte, 12 cycle	8-byte, 18 cycle

۲-۵- 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 دسته-های 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[۱] جریان بسته‌های TCP/IP و سایر دستکاری‌ها روی بسته‌ها می‌باشد.

۱ defragmentation

فـصل ۳ :

مروری بر کارهای مرتبط

۳-۱- مقدمه

همانطور که پیش‌تر گفته شد، یکی از پارامترهای مهم در طراحی سیستم‌های تعبیه شده میزان توان مصرفی در آن‌هاست. در این پروژه نیز هدف بر این است که برنامه‌های بسته محک MiBench که یک بسته استاندارد برای کاربردهای تعبیه شده است بر روی یک سیستم تعبیه شده مبتنی بر ARM اجرا شده و میزان توان مصرفی آن‌ها به ازای قسمت‌های مختلف برنامه استخراج گردیده و تحلیل 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- سخت-افزاری 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.

۳-۲- آشنایی با شبیه‌ساز MEET[۱]

۳-۲-۱- 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"[۲] 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

۱ Microcontroller Energy Estimation Tool

۲ „An Accurate Instruction-Level Energy Estimation Model and Tool for Embedded Systems“

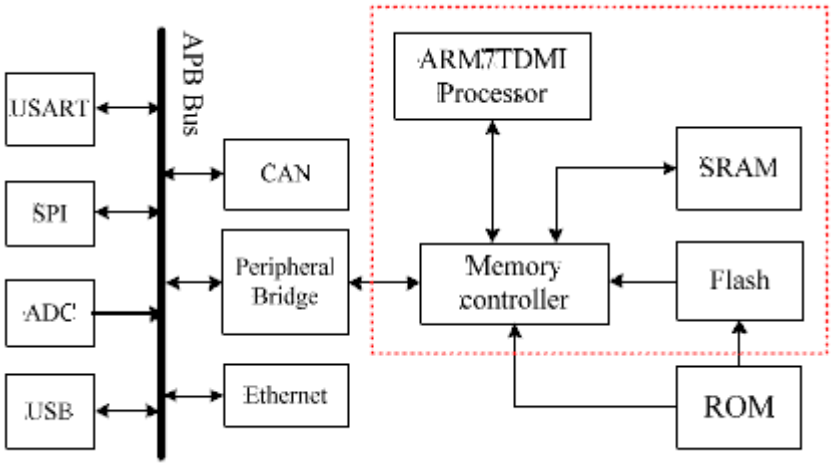
شبیه‌سازی می‌کند. کار با MEET بسیار شبیه به Sim-profile است و هرکسی با دانش اندکی از نحوه‌ی ساخت و اجرای پروژه در شبیه‌ساز Simplescalar، برای کار با MEET مشکلی نخواهد داشت. برای ساده کردن فرآیند کامپایل و اجرای برنامه‌ها، سه پوسته‌ی اسکریپت به بسته اضافه شده‌اند: build.sh، extractor.sh و run.sh. build.sh 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.

۳-۲-۲- بستر سخت‌افزاری

بستر سخت‌افزاری شبیه‌سازی شده توسط MEET، یک میکروکنترلر AT91SAM7X256 است که با ۶۴ کیلوبایت حافظه‌ی رم استاتیک، ۲۵۶ کیلوبایت حافظه فلش و یک پردازنده‌ی ARM7TDMI مجهز شده است. ساختار داخلی این میکروکنترلر در شکل ۳-۱ 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.



شکل (۳-۱) بستر سخت‌افزاری شبیه‌سازی شده توسط MEET

MEET بین دسترسی به حافظه‌ی فلش و دسترسی به حافظه‌ی رم استاتیک بـه وسـیله‌ی مقـدار آدرس هدف تمایز ایجاد می‌کند. حافظه‌ی فلـش از آدرس 0x100000 شـروع مـی‌شـود در حـالی‌کـه حافظـه‌ی رم استاتیک از آدرس 0x200000 شروع می‌شود. بنابراین برای رسیدن به نتایج دقیق، بایـد اسـکریپت لینکـر را با توجه به تنظیمات حافظه تغییر داد.

۳-۲-۳- سیستم مورد نیاز

سیستم مورد نیاز برای MEET بسیار شبیه به سیمپل‌اسکالر است. MEET را می‌توان به وسـیله‌ی کامپـایلر GCC در محیط لینوکس کامپایل کرد. ورژن 1.1 با استفاده از تنظیمات زیر آزمایش شده است، امـا سـاخت 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

۳-۲-۴- 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 عنـوان مثـال دسـتور #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 مقداردهی-های اولیه هم می‌باشد را برآورد کنیم).

- sh run.sh /home/mostafa/source/quicksort main main

۳-۲-۵- گزینه‌ها[۱]

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 ۳ آپشن جدید وجود دارد که در جدول زیر لیست شده است.

Option	Purpose
-initial:pc [address]	Start the execution from specific address. If this option is not specified, the starting address of program from the binary image is used instead.
-finish:pc [address]	End the execution after reaching the specific address.
-initial:meas [address]	Start the estimation process after reaching the specified address. If this option is not specified, the estimation process will start from the beginning of the application.

شکل (۳-۲) گزینه‌های جدید MEET

۳-۲-۶- مشخصه‌های شبیه‌سازی[۲]

مشابه قسمت قبل، نتایج نهایی شبیه‌سازی در MEET به جز در ۷ مورد شبیه Sim-profile است. این ۷ مورد در جدول زیر لیست شده‌اند.

---

۱ Options

۲ Simulation statistics

Option	Purpose
inst_count_after_meas	Total number of instructions executed after starting of estimation procedure.
sim_num_flash_loads	Total number of Flash read memory accesses
sim_num_sram_loads	Total number of SRAM read memory accesses
sim_total_energy	Total energy consumption of the simulation (nJ)
instruction_bus_activity	Total number of bit flips in instruction bus
instruction_bus_weight	Total number of '1' bits in instruction bus
regbank_activity	Total number of bit flip in register bank

شکل (۳-۳) مشخصه‌های جدید شبیه‌سازی

۳-۲-۷- 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 "۱" 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

فـصل ۴ :

شبیه ساز MEET

۴-۱- 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[۱] 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.

۴-۲- شرح برنامه‌ها و نتایج شبیه‌سازی آن‌ها

۴-۲-۱- Bitcount

همان‌طور که پیش از این گفته شد، این الگوریتم قابلیت دستکاری بیت‌ها در یک پردازنده را به وسیله‌ی

---

۱ Hard-Code

شمارش تعداد بیت‌ها در یک آرایه از اعداد صحیح تست می‌کند. این فرآیند از طریق پنج روش قابل انجام است:

۱) شمارنده‌ی بهینه‌شده‌ی یک بیت در هر حلقه[۱]
۲) شمارش بازگشتی ۴ بیتی[۲]
۳) شمارش غیر بازگشتی ۴ بیتی با استفاده از جدول جستجو[۳]
۴) شمارش غیر بازگشتی ۸ بیتی با استفاده از جدول جستجو[۴]
۵) شیفت دادن و شمارش بیت‌ها[۵]

در الگوریتم شماره‌ی ۱ (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 ۲ که توسط رتکو تامیک[۶] نوشته شده است، به صورت بازگشتی شمارش بیت‌ها را به صورت ۴ بیتی انجام می‌دهد.

الگوریتم شماره ۳ که توسط باب استوت[۷] نوشته شده است، با استفاده از یک جدول جستجو شمارش غیر بازگشتی را انجام می‌دهد. نکته‌ای که وجود دارد این است که ۱۶ ورودی اول جدول برای این الگوریتم استفاده می‌شود. مابقی می‌توانند حذف شوند.

الگوریتم شماره ۴ که توسط بروس ودینگ و اوک ریتسما[۸] نوشته شده است، همانند الگوریتم ۳ است، با این تفاوت که شمارش به صورت ۸ بیتی انجام می‌شود.

در الگوریتم شماره ۵ 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 '۰' و '۱' است.

---

۱ Optimized 1 bit/loop counter
۲ recursive bit count by nibbles
۳ non-recursive bit count by nibbles using a table look-up
۴ non-recursive bit count by bytes using a table look-up
۵ Shift and count bits
۶ Ratko Tomic
۷ Bob Stout
۸ Auke Reitsma and Bruce Wedding

نتایج شبیه‌سازی:

نتایج شبیه‌سازی برای الگوریتم Bitcount به شرح زیر است:

جدول (۴-۱) 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

۴-۲-۲- دیکسترا

این الگوریتم یکی از الگوریتم‌های پیمایش گراف است که مسئله‌ی کوتاه‌ترین مسیر از مبدأ واحد را برای گراف‌های وزن‌داری که یال با وزن منفی ندارند، حل می‌کند و در نهایت با ایجاد درخت کوتاه‌ترین مسیر، کوتاه‌ترین مسیر از مبدأ به همه‌ی رأس‌های گراف 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 انتخاب شده است.

روند الگوریتم دیکسترا مطابق زیر می‌باشد:

۳) انتخاب راس مبدا

۴) 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.

۵) راس مبدا با اندیس صفر را در داخل S قرار می‌دهد.

۶) 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.

۷) از رئوس خارج مجموعه، راسی با کمترین اندیس انتخاب شده و به مجموعه‌ی S اضافه می‌گردد.

۸) این کار را دوباره از مرحله‌ی ۴ 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 ۵۰×۵۰:

جدول (۴-۲) 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 ۱۰۰×۱۰۰:

جدول (۴-۳) 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

Stringsearch – ۳ – ۲ – ۴

همان‌طور که پیش از این گفته شده بود، این محک با استفاده از الگوریتم مقایسه‌ی حساس به حروف[۱] در عبارات به دنبال کلمات داده شده می‌گردد.

این الگوریتم در واقع بخشی از الگوریتم جستجوی رشته‌ی پرات- بویر- مور[۲] است که در سال ۱۹۹۱

۱ case insensitive comparision algorithm

۲ Pratt-Boyer-Moore string search

توسط جری کافین نوشته شده است. این بخش از الگوریتم کـه در Mibench اسـتفاده شـده، در اوایـل سـال ۱۹۹۱ از برنامه ی اصلی جدا شد و با بازنویسی جداگانه‌ی آن، برای استفاده‌ی عمومی آمـاده شـد. سـپس در اواخر مارس و اوایل آپریل با کمک تاد اسمیت[۱] اصلاح شد.

در این الگوریتم تابع Init_search به همراه رشته ای که شامل کلمه‌ی مـورد نظـر بـرای جسـتجو اسـت، برای قرار گرفتن در جدول مقدار اولیه فراخوانی می‌شود. سپس تـابع Strsearch بـه همـراه یـک بـافر بـرای 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 ۶۰ رشته:

جدول (۴-۴) 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

۱ Thad Smith

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 ۱۳۰۰ رشته:

جدول (۴-۵) 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

۴-۲-۴- بلوفیش

این الگوریتم یک رمزنگار بلوکی است که بر روی مقادیر ۶۴ بیتی (۸بایتی) عملیات انجام می‌دهد. این الگوریتم از مقادیر مختلف کلید برای رمزنگاری می‌تواند استفاده کند ولی عموما مقدار ۱۲۸ بیتی (۱۶بایتی) 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 ۱۶ دور دارد. هر دور شامل یک جایگشت وابسته به کلید و یک جانشانی وابسته به کلید و داده‌است. تمامی عملگرها XORها 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 ۶۴ 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 ۸ 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 ۲۳۰۰۰۰ کاراکتر:

جدول (۴-۶) 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 ۴۶۰۰۰۰ کاراکتر:

جدول (۴-۷) 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

Rijndael – ۵ – ۲ – ۴

الگوریتم Rijndael که یک الگوریتم رمزنگاری با گزینه‌های ۱۲۸، ۱۹۲ و ۲۵۶ 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[۱] و وینسنت ریجمن[۲] 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 ۰ یا ۱ بلوک) دارند، "i+1" برای رمزنگاری و "14-i" به عنوان بافر در نظر گرفته می‌شود.

◘ طرز کار الگوریتم

الگوریتم Rijndael بایت به بایت کار می‌کند و ورودی اصلی را با کلید رمزنگاری در یک ماتریس ۴×۴ جفت می‌کند. کلید، به طریقی تقسیم یا برنامه‌ریزی شده است که بتواند در مراحل مختلف تکرار به تدریج تزریق شود. اولین قسمت کلید قبل از شروع پروسه ۱۰ مرحله‌ای تزریق می‌شود. در هر کدام از این مراحل، بایت‌ها جابجا می‌شوند، ردیف‌ها شیفت پیدا می‌کنند و ستون‌ها ترکیب می‌شوند.

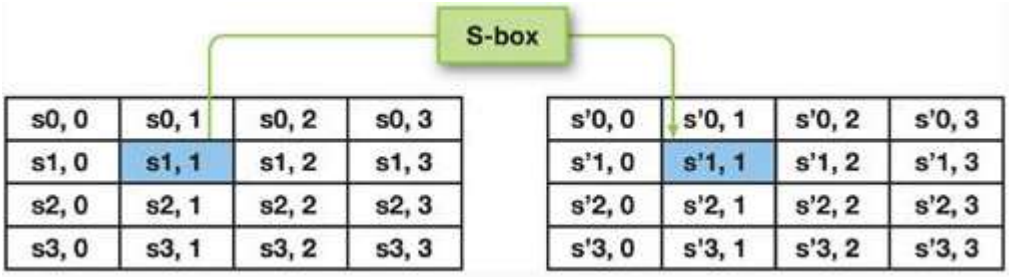
- **SubBytes**

در پروسه جابجایی، بایت‌های متن ورودی در یک جعبه جابجایی به نام S-box قرار می‌گیرند که یک

۱ Joan Daemen

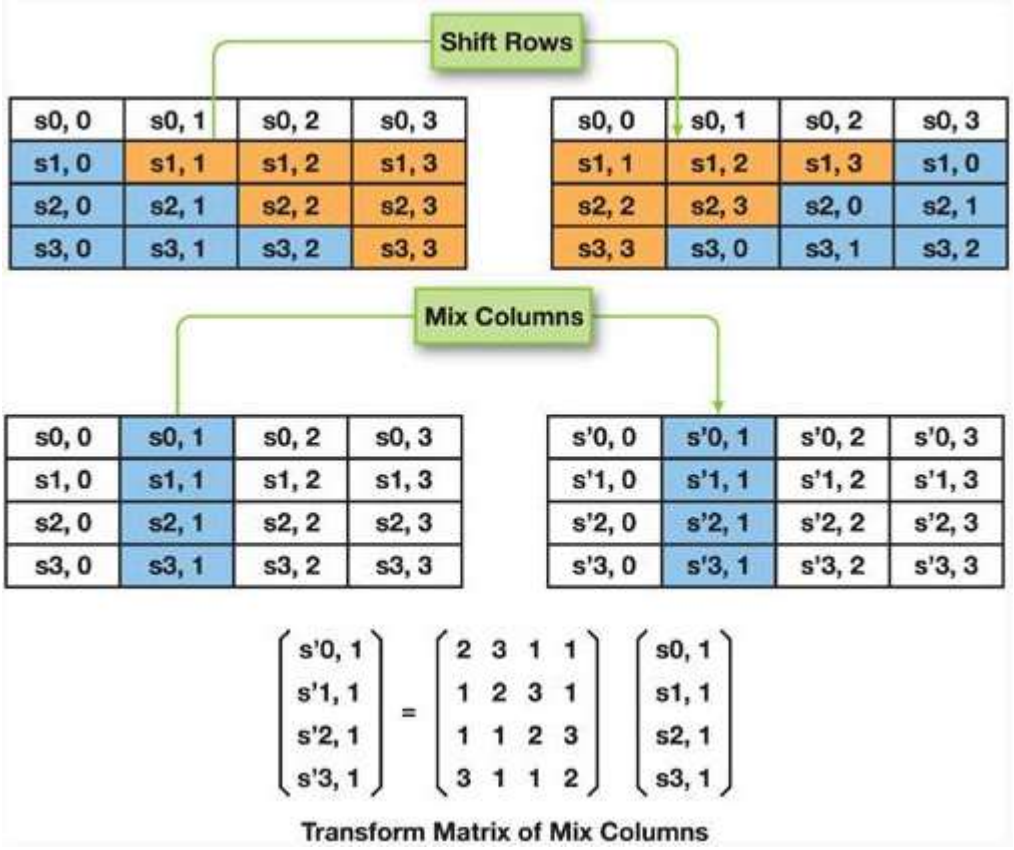
۲ Vincent Rijmen

ماتریس ۱۶×۱۶ است. هر بایت در یک تقاطع سطر و ستون این ماتریس جا می‌گیرد. برای پیدا کردن جای هر بایت اولین عدد صحیح مبنای ۱۶ (nibble) در یک بایت متن اصلی گرفته شده و از آن برای مشخص کردن سطر S-box استفاده می‌شود و سپس از دومین nibble برای مشخص کردن ستون استفاده می‌شود. کاراکتری که در تقاطع سطر و ستون انتخاب شده ذخیره می‌گردد، به عنوان SubByte برای متن اصلی شناخته می‌شود. این پروسه برای هر ۱۶ بایت در ماتریس تکرار می‌شود.



- **شیفت ردیف و ترکیب ستون‌ها**

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



Transform Matrix of Mix Columns

- **کلیدهای Round**

در مرحله بعدی یک کلید Round به هر ستون اضافه می‌شود. این کلید در واقع یک تکه کوچک از یک کلید محرمانه است که برای مراحل بعدی رمزنگاری تزریق می‌شود.

- **تکرار**

این تبدیل‌ها ۹ بار دیگر تکرار می‌شوند. در تکرار آخر ترکیب ستون ها وجود ندارد و با اضافه کردن کلید Round متن رمزنگاری شده به دست می‌آید. کلید نیز به نوبه خود شیفت پیدا می‌کند، 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 ۲۳۰۰۰۰ کاراکتر:

جدول (۴-۸) 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 ۴۶۰۰۰۰ کاراکتر:

جدول (۴-۹) 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

Sha – ۶ – ۲ – ۴

یک الگوریتم درهم سازی امن است که یک پیام خلاصه‌ی ۱۶۰ 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[۱] 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 ۲۳۰۰۰۰ کاراکتر:

جدول (۴-۱۰) 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

۱ National Institute of Standards and Technology

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 ۴۶۰۰۰۰ کاراکتر:

جدول (۴-۱۱) 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

فـصل ۵ :

ارزیابی نتایج شبیه سازی

۵-۱- 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.

۵-۲- ارزیابی نتایج شبیه سازی الگوریتم‌های مختلف

۵-۲-۱- دیکسترا

۱- برای ورودی آرایه‌ای با ابعاد ۵۰ ×۵۰:

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

۲- برای ورودی آرایه‌ای با ابعاد ۱۰۰×۱۰۰:

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 ۴ برابر شدن سایز ورودی، تعداد دستورالعمل‌ها تقریبا ۱۴ برابر شده است. انرژی مصرفی ۱،۰۱ برابر شده و زمان شبیه‌سازی نیز ۱۵،۹۸ برابر شده است. بنابراین توان مصرفی در این الگوریتم با دو برابر شدن ورودی ۰،۰۶ برابر شده است. این بدین معنی است که با افزایش سایز ورودی توان مصرفی به میزان قابل توجهی کم می‌شود.

۵-۲-۲- Stringsearch

۱- برای ورودی کوچک با سایز تقریبی ۶۰ 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

۲- برای ورودی بزرگ با سایز تقریبی ۱۳۰۰ 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 ۲۲ برابر شدن سایز ورودی، تعداد دستورالعمل‌ها تقریبا ۱۰ برابر شده است. انرژی مصرفی ۱۴,۲ برابر شده و زمان شبیه‌سازی نیز ۹ برابر شده است. بنابراین توان مصرفی در این الگوریتم با دو برابر شدن ورودی ۱،۵۸ برابر شده است. این بدین معنی است که با افزایش سایز ورودی توان مصرفی به میزان قابل توجهی زیاد می‌شود.

۵-۲-۳- بلوفیش

۱- برای ورودی کوچک با سایز تقریبی ۲۳۰۰۰۰ 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

۲- برای ورودی بزرگ با سایز تقریبی ۴۶۰۰۰۰ 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.

Rijndael – ۴ – ۲ – ۵

۱- برای ورودی کوچک با سایز تقریبی ۲۳۰۰۰۰ کاراکتر:

sim_num_insn	**473197189 # total number of instructions executed**
sim_total_energy	134521840.0000 # total energy consumption (nJ)
sim_elapsed_time	513total simulation time in seconds

۲- برای ورودی بزرگ با سایز تقریبی ۴۶۰۰۰۰ 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 ۱،۰۳ برابر شده و زمان شبیه‌سازی نیز ۱،۳۵ برابر شده است. بنابراین توان مصرفی در این الگوریتم با دو برابر شدن ورودی ۰،۷۶ برابر شده است. این بدین معنی است که با افزایش سایز ورودی توان مصرفی به مرور کم‌تر می‌شود.

Sha – ۵ – ۲ – ۵

۱- برای ورودی کوچک با سایز تقریبی ۲۳۰۰۰۰ 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

۲- برای ورودی بزرگ با سایز تقریبی ۴۶۰۰۰۰ 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 ۱،۴ برابر شده و زمان شبیه‌سازی نیز ۱،۶۷ برابر شده است. بنابراین توان مصرفی در این الگوریتم با دو برابر شدن ورودی

۰،۸۵ برابر شده است. این بدین معنی است که با افزایش سایز ورودی توان مصرفی به مرور کم‌تر می‌شود.

۵-۲-۶- مقایسه‌ی الگوریتم‌های بلوفیش و Rijndael

۱- برای ورودی کوچک با سایز تقریبی ۲۳۰۰۰۰ 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 ۵،۶ برابر Rijndael است. لذا توان مصرفی کل برای الگوریتم Rijndael، ۵،۶ برابر توان مصرفی الگوریتم بلوفیش به ازای ورودی یکسان است.

۲- برای ورودی بزرگ با سایز تقریبی ۴۶۰۰۰۰ 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، ۱،۰۳ برابر الگوریتم بلوفیش است، و زمان اجرای الگوریتم بلوفیش ۸،۱۷ برابر Rijndael است. لذا توان مصرفی کل برای الگوریتم Rijndael، ۸،۴۱ برابر توان مصرفی الگوریتم بلوفیش به ازای ورودی یکسان است.

فـصل ۶ :

نتیجه‌گیری و کارهای آینده

۶-۱- نتیجه‌گیری

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 تغییراتی در نحوه‌ی ورودی گرفتن برنامه‌ها آن‌ها را برای شبیه‌سازی آماده کردیم و با استفاده از شبیه-ساز 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.

۶-۲- کارهای آینده

در خصوص کارهایی که که می‌توان در ادامه و با استفاده از نتایج 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 فارسی به انگلیسی

آمار شبیه‌سازی ..... Simulation Statistics
اتوماسیون اداری ..... Office Automation
اتوماسیون و کنترل صنعتی..... Automotive and Industrial Control
ارتباطات ..... Telecommunications
امضای دیجیتال ..... Digital Signature
امنیت ..... Security
انرژی مصرفی ..... Energy Consumption
انشعاب..... Branch
بازگشتی..... Recursive
بستر نرم‌افزاری ..... platforme
بلافصل ..... Immediate
بهینه شده ..... Optimized
تخمین..... Estimation
توان مصرفی..... Power Consumption
خط لوله..... Pipeline
دستکاری..... Manipulation
رمزگذاری ..... Encrypt
رمزگشایی..... Decrypt
سیستم تعبیه‌شده ..... Embeded system
سیستم رومیزی..... Desktop system
سیستم میزبان ..... Host system
شبکه..... Networking
شبیه‌سازی..... Simulation
شرکت پذیری ..... Associative
غیر بازگشتی..... Nonrecursive
قابل حمل..... Portable
قطعات مصرف کننده ..... Consumer Devices
کارایی..... Performance
کامپیوترهای همه منظوره..... General-purpose computers
متقارن ..... Symmetric
محک..... Benchmark
معتبرسازی ..... Validation

ممیز شناور ..... Floating point
موازی‌سازی ..... Parallelism
نامتقارن ..... Asymmetric
نیم‌بایت ..... Nibble
یکپارچه‌سازی ..... Defragmentation

بخش ب: واژه نامه انگلیسی به فارسی

نامتقارن ..... Asymmetric
شرکت پذیری ..... Associative
اتوماسیون و کنترل صنعتی..... Automotive and Industrial Control
محک..... Benchmark
انشعاب..... Branch
قطعات مصرف کننده ..... Consumer Devices
رمزگشایی..... Decrypt
یکپارچه‌سازی..... Defragmentation
سیستم رومیزی..... Desktop system
امضای دیجیتال ..... Digital Signature
سیستم تعبیه‌شده ..... Embeded system
رمزگذاری ..... Encrypt
انرژی مصرفی ..... Energy Consumption
تخمین..... Estimation
ممیز شناور ..... Floating point
کامپیوترهای همه منظوره..... General-purpose computers
سیستم میزبان ..... Host system
بلافصل ..... Immediate
دستکاری..... Manipulation
شبکه..... Networking
نیم‌بایت..... Nibble
غیر بازگشتی ..... Nonrecursive
بهینه شده ..... Optimized
آمار شبیه‌سازی ..... Simulation Statistics
اتوماسیون اداری ..... Office Automation
موازی‌سازی ..... Parallelism
کارایی..... Performance
خط لوله..... Pipeline
بستر نرم‌افزاری ..... Platforme
قابل حمل..... Portable
توان مصرفی..... Power Consumption
بازگشتی..... Recursive

امنیت ..... Security
شبیه‌سازی ..... Simulation
متقارن ..... Symmetric
ارتباطات ..... Telecommunications
معتبرسازی ..... Validation

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